UNIVERSITY OF OSLO
Department of Informatics

Towards Automated Transformation of UML/OCL to Prolog for Efficient Configuration of Integrated Control Systems

Thomas Gramstad Rolfsnes

1. August, 2013

Abstract

The configuration of product-lines can be an error-prone and time consuming process. In this thesis we extend on the work by Behjati [A model-based approach to the software configuration of integrated control systems,2012], where a product-line methodology has been proposed. The end goal is to implement a configuration tool that will ease the product-line configuration process. The configuration tool demands two model transformations. The first transformation produces an intermediate model (used internally by the tool) from a product-line model, the implementation of this transformation is fully described in this thesis. The second transformation uses this intermediate model, together with OCL constraints, to produce a logic-program, specified in Prolog. The efficiency of the generated Prolog code is critical for the configuration tool. In this thesis we take steps towards this transformation. We hypothesized that the following aspects of the Prolog code is critical for efficiency: How instances of the intermediate model are represented in Prolog (i.e., the structure of the Prolog query), how associations are represented and resolved, and how the Prolog predicates, representing the intermediate-model and OCL, are organized. We performed a large scale experiment investigating these factors, as well as the impact of instance size, the configuration of attributes (i.e., if values has been set), and impact of changing the OCL constraint sets used. The main findings were the following: Representing the instances as a binary-tree structure, combined with a resolution of associations through id-references, yielded the highest efficiency. Further, condensing the Prolog predicates led to increased efficiency in cases where the individual predicates (i.e., the transformed OCL constraints), shared association-navigations. The configuration of attributes were found to not impact efficiency. Future work can take advantage of these findings to move even closer to getting a fulfilled and efficient transformation to Prolog.

Aknowledgments

First and foremost I want to thank my supervisors at Simula, Razieh Behjati and Tao Yue. I also would like to thank my internal supervisor at IfI, Magne Jørgensen (especially for sending me a copy of his signature from China!). Me and Razieh has spent countless hours in discussion, and exchanged over 100 e-mail over the last 12 months (I counted). Her guidance has been invaluable.

I would also like to thank my family and friend for their motivation and support. Last, but not least, I must extend a large thank you to my girlfriend Ingrid. I promise I'll make all the dinners the next 12 months!

Contents

Chapter 1

Introduction

The increasing use of embedded software systems in all aspects of our lives – e.g., in home appliances, in our cars, and in complex industrial machines – demands for high-quality and low-cost software products. To improve quality and to reduce production costs, many organizations have adopted software product-line engineering approaches ([1] [2] [3]) to develop the software embedded in their systems. These product lines typically consist of a large variety of reusable components. Software development, in this context, involves selecting and customizing the reusable components according to the specific needs of a particular customer. We refer to this as the *configuration* process.

In many industrial contexts, due to their complexities, the configuration process becomes time consuming and error prone [4]. In [5], a model-based, semi-automated configuration framework has been proposed to overcome such configuration challenges. Figure 1.1 shows an overview of the framework and its major parts. Two model transformation steps are contrived to provide the required end-to-end automation while making the framework independent from the input notation (i.e., model of the product family).

This thesis investigates the two model transformation steps needed in the framework. The first one is a model-to-model transformation to transform SimPL-models of the product family into an intermediate model used by the configuration engine. Here we report the details of our implementation, and provide an example transformation of a SimPL-model. The second model-transformation step will in the end generate a logic program from the aforementioned intermediate model of the product family, and the constraints defined in it. This logic program, which is specified in Prolog/clpfd, is used by the configuration engine for the purpose of configuration validation and guidance generation. This thesis describes the steps we have taken to approach such an efficient logic program. We experiment with several different mappings between the intermediate model and the logic program, and report our findings. The goal is that these findings will guide future implementations of the transformation, such that the resulting logic program is efficient.

1.1 SimPL

The Simula Product Line (SimPL) methodology, was conceived in an effort to overcome challenges related to the configuration of families of Integrated Control Systems (ICS) [4]. ICSs can

be described as *"[...] heterogeneous systems-of-systems, where software and hardware components are integrated to control and monitor physical devices and processes [...]"*[4]. A family of ICSs are simply ICSs that share a common code base. Working with such artifacts is often termed Product-line engineering. One can think of a product-line being the family of ICSs, and a *product* being an instance of the ICS family. In Model Driven Engineering (MDE) , a product-line is represented through a generic model. To enable the generation of different products, several *variation points* are introduced by the model[6]. The process of instantiating all these variation points is called *configuration*. The configuration is done iteratively. In each iteration the user provides a configuration decision, and the configuration tool provides feedback (i.e., provides guidance and inference). This process however, can often be quite complicated and error-prone, due to the interdependencies that exist between variation points [4]. This is where SimPL enters, by providing a methodology, along with a UML-profile, for creating product-line models. Ultimately, these models will act as input to a configuration tool, in which the goal is to aid end-users throughout the configuration process. User assistance is achieved through providing guidance (i.e. what are the valid instantiations of the variation point?), and through inferring the valid instantiation of variation points. This is possible through leveraging information from the model (multiplicity of associations, etc.), but more prominently through the Object Constraint Language (OCL) . In fact, OCL is used to describe the interdependencies between variation points, resulting in a narrowing of the legal instantiations of the model. OCL will be thoroughly discussed in section 2.2.

The main focus of this thesis however, is not ICSs, SimPL or product-lines, they merely provide the backdrop of our problem domain. The main focus revolves around the scalability of a model-transformation, needed within a large-scale system.

1.2 Steps to get a working configuration tool

An overview of the configuration-tool framework, presented in [7], can be found in Figure 1.1. The diagram has been annotated with the parts that will be discussed in this thesis. In particular, two transformations are needed. The first transformation is annotated at "2", while the second is split in two parts, at "3" and "4".

a.x: Annotation x, referring to the corresponding number in Figure 1.1.

a.1: A product-line model (i.e. a SimPl-model), with attached OCL constraints.

a.2: The first transformation step is from a SimPL-model to an intermediate model, specifically designed for the configuration process. An explanation of the target model, along with the implemented transformation, can be found in chapter 3. We have simply named this tool specific model, the *Tool-model*

a.3: Part one of the second transformation step. The OCL and (parts of the) Tool-model are transformed to a set of Prolog predicates, which will be used to provide guidance and infer new variation point instantiations. For each product-family model, this transformation is done only once.

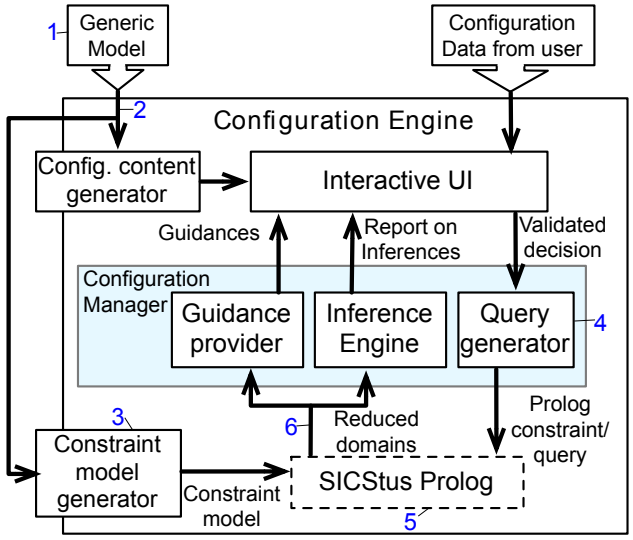

Figure 1.1: Overview of the configuration tool framework.

a.4: Part two of the second transformation step. At the end of each configuration iteration, a Prolog query is created from the current state of the system-under-configuration. This is an instance of the Tool-model, and we will from now on refer to it as the Tool-instance[1]. This Prolog query is passed to the SICStus Prolog engine to be evaluated using the Prolog predicates generated in step a.3. The result of evaluation forms the guidance and inference that is presented to the user. Note that this transformation is repeated at the end of each configuration iteration, until the process is completed, *or* until a complete configuration is achieved.

a.5: Through the use of *Jasper* [8], we are able to interface with a SICStus Prolog engine, which again is used to evaluate the query. Evaluating the query over the Prolog predicates can be represented as a Constraint Satisfaction Problem (CSP), which is used to provide guidance and infer the resolution of variation points. CSPs and SICStus Prolog are discussed in section 2.4.

a.6: These components of the configuration tool convert the results of the Prolog evaluation into a easily comprehensible format for the end-user.

If one is interested in the details of the rest of the configuration engine, we refer to [7].

1.3 Transformation overview

In Figure 1.2, we have extracted the two steps of transformation from Figure 1.1.

The first transformation step, SimPL2Tool, has been fully implemented, and is discussed in chapter 3. The Tool-model merely functions as an intermediate model, where the relevant data for configuration, has been extracted from the SimPL-model. The second transformation step,

[1]The Tool-instance is currently realized in Java.

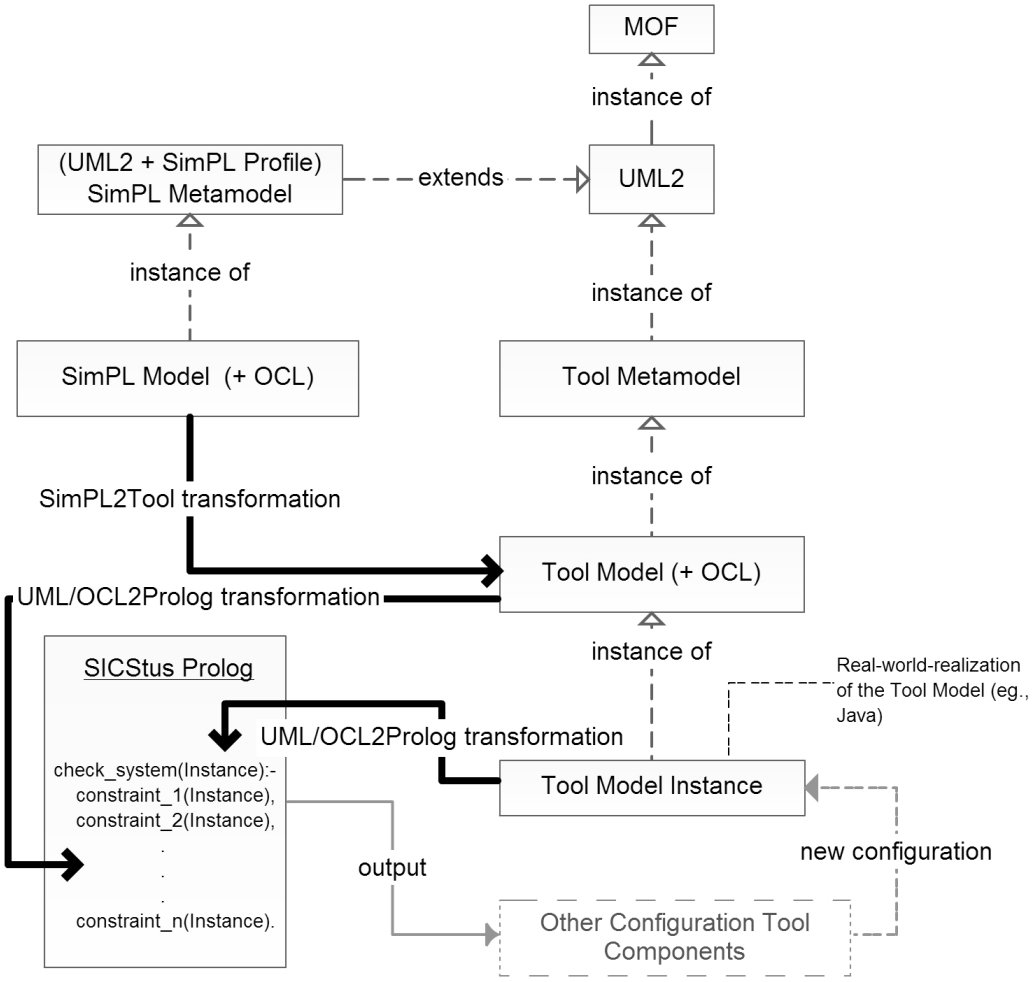

Figure 1.2: Overview of transformations and models, and their place in the configuration process.

UML/OCL2Prolog, is really two different transformations. But we have kept them under the same umbrella for simplicity. The first part, is a transformation from two sources: A set of OCL constraints, plus the Tool-model. Here the target is a set of Prolog predicates, which should encapsulate the same combined constraints as the two sources. It is important to note that this part of the transformation will only happen once for each product-family model. The second part, is the transformation from a Tool-instance, to a Prolog query (note that the query here is represented by the Prolog variable; "Instance"). The Tool-instance represents the current state of the configuration: the number of objects, associations between them, and if attributes has been configured yet, etc. The transformed query will encapsulate this same information, and is then evaluated over the previously generated predicates. This part of the transformation will happen numerous times through the configuration process.

Now, as seen in the title of this thesis, we are not only interested in these transformations, but in the efficiency of the resulting Prolog code. Indeed, most of this thesis will have efficiency as its focus. By looking at Figure 1.1, the reason should be clear. How fast the Prolog code runs (i.e., the efficiency), will directly impact the end-user, and is critical for successful industry adoption of our configuration tool. Anything more than a few seconds evaluation time, is not a pleasant user experience.

1.4 Aiming at efficiency

Firstly, to be entirely clear, by efficiency we are referring to the minimization of Prolog Execution Time (PET) [2]. The observant reader might have also noticed that there is a transformation step within the configuration-process-loop, namely from the Tool-instance to the Prolog query. The execution time here, is of course also of great importance. It will not however, be discussed in this thesis. The main reason being that we do a lot of extra calculation in the query generation, related to the setup of our experiment. Hence we leave the investigation of efficiency in this transformation step to later work.

To achieve our goal, we hypothesized a set of *choice-points* in the UML/OCL2Prolog transformation. We define a choice-point as some part of the transformation, where several mappings exist. Moreover, we believed that choosing one mapping over the other, might have an impact on PET. To test the impact on PET of these choice points, a small SimPL-model, and two sets of OCL constraints were conceived. These laid the groundwork for a large scale experiment, where all the combinations of choice-points where tested for impact on PET. Further, things like model-size and type of attached-constraint were also included in the experiment to see how these affected PET. We will refer to these, and the choice-points, as the *factors* in our experiment. Moreover, the variation within each factor will be referred to as the *"factor-levels"*. In total, 96 different combinations of factor-levels existed, and were replicated 10 times each to be able to measure the mean PET. By calculating the mean, we were able to say something about the significance (i.e., if it had impact on PET) of each factor. This were done through a statistical method called Analysis of Variance (ANOVA). The preliminary results can be found in section 5.4, and analysis using ANOVA can be found in section 6.2.

1.5 The question of complexity

In [9], a method for calculating the *complexity* of OCL constraint were proposed. Here, the complexity is defined as the number of objects that must be considered to be able to evaluate the constraint. We found that this complexity calculation is dependent on the nature of several of the factors in our experiment. We therefore hypothesized that the complexity in it self can be a predictor of PET. This hypothesis will be investigated alongside our main experiment.

1.6 Manual vs automatic transformation

The first transformation step, discussed in chapter 3, has a nearly completed implementation in ATL [10]. We are aware of a few faulty mappings, and especially *enumerations* are not properly handled at this point. We are confident these issues will be resolved in the future. However, much because of this, we will not use this ATL implementation in relation to our experiments with the next transformation step. In fact, we are not using the SimPL and Tool-models "directly" (see Figure 1.2). Instead we have carefully, manually transformed the SimPL-

[2] We have tried to be consistent with the word use here. Still, there might be instances where we use variations of 'performance'. They should all be interpreted as refering to Prolog execution time (PET). Efficient = low PET = high performance. Mostly we will use high/low PET.

model presented in subsection 2.3.1 to a Tool-model. Taking care that we at all times are conforming to the respective meta-models. The second transformation step is performed in a similar fashion. The various Tool-instances used in our experiment, are generated as Java objects, and transformed to Prolog queries using PQG[3] (see section 5.3). OCL constraints were manually implemented as Prolog predicates, any needed information from the Tool-model were also handled manually (e.g., the handling of enumerations). Another important aspect of the Tool-model, is the constraints that stem from multiplicity on associations. However, as will be discussed later, we are only operating with multiplicities of *1* in this thesis. This of course significantly reduces the amount of information that is needed from the Tool-model. All of these issues should be addressed in future work.

1.7 Outline

In chapter 2, we start by a brief introduction to the parts of the UML that are relevant in our context. This is followed by a similar introduction to OCL. Particularly, we look at the semantics of the OCL operations that later will be implemented/transformed to Prolog. Next, we discuss the SimPL-profile. The section rounds off by an introduction to the calculation of OCL constraint complexity. Finally, we explain how the configuration process can be seen as a series of Constraint Satisfaction Problems (CSPs), and how they can be solved in SICStus Prolog using the module: 'Constraint Logic Programming for Finite Domains' (clpfd) [11]. In chapter 3 we look at the mapping rules that have been written for the SimPL2Tool transformation, and how they were implemented in the Atlas Transformation Language (ATL). Then in chapter 4, we look at the UML/OCL2Prolog transformation where we enumerate the different mapping choices, and their implementations in Prolog. Further, we see how the OCL constraints chosen for the experiment were implemented as Prolog predicates. In chapter 5, we start the preparation for the experiment with a reiteration of all factors. Next we explain how these factors were combined to form a full factorial design, which makes the experiment-data suitable for analysis using the statistical method: Analysis of Variance (ANOVA) [12]. The chapter rounds off with an enumeration of all preliminary experiment results. In chapter 6 we analyze the results, with the goal of identifying the factors that had a significant effect on PET. Further, we try to find the combination of factor-levels that yielded the lowest mean PET. In addition, we analyze the results of the separate OCL constraint complexity experiment, to determine if it indeed is a good predictor of PET. The chapter rounds off with a discussion of our method, with a focus on the validity of our results. In chapter 7 we look at some related work, and the implications it might have for further research. Then in chapter 8, we present concluding remarks.

[3]This is a small Java library we developed to run our experiment.

Chapter 2

Background

2.1 UML

The Unified Modeling Language (UML) is a *"general-purpose visual modeling language for systems"*[13]. It provides different types of diagrams, both for modeling *structure*, and *behaviour* of a system. In our context we only consider the structure diagram, namely the *class-diagram*. Models here, can exist on different abstraction layers. In Figure 1.2, we've tried to topologically illustrate which abstraction layer each model resides in. On the top, we have the Meta Object Facility (MOF). MOF is the meta-model for the layer below, which for us is UML2. The SimPL meta-model also resides at this level, and is a special instance of UML2, which has been enhanced with the SimPL profile. Profiles will be discussed later. In essence, a meta-model provides the constructs necessary for defining models on the abstraction layer below [14]. That model can act as the meta-model for the layer below that and so on. So there is no topological restriction on where the meta-models reside, and where the models reside. This brings us to *our* bottom layer[1], which also can be called the *data-layer*. Here we have the *real-world-realizations*[2] of the system[14]. In our context, we have two different representations at this layer. The Tool-instance (i.e., realization in Java), and its representation as a Prolog query[3].

2.2 OCL

In this section we explain the need for the OCL in a MDE context. We go into detail on some OCL language components that have been used in the experiment. Then EOS[16] is presented as a means for parsing OCL constraints attached to a model. Finally we discuss how OCL constraint complexity can be understood, together with algorithms for generating complexity functions for constraints.

[1] The numbers of possible layers are unrestricted [15]. This is just the number of layers we have.

[2] Also referred to as the run-time instances.

[3] In later chapters, especially chapter 4, "Tool-instance" is also used to refer to the contents of the input Prolog query

2.2.1 OCL in MDE

While UML class diagrams enables the modeling of the structure of a system, there still often exists other aspects that are not captured. Consider the simple example in Figure 2.1. Although it is natural to represent the age attribute of a person as an integer, the possible value range does not translate well to the possible life span of a person. A persons age can never be less than 0, and it might also be natural to set a max bound of lets say 130[4]. OCL came out of the need to also be able to put such constraints upon models.

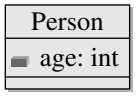

Figure 2.1: Can a person have negative age? Or live a 1000 years?

For our Person example, we could write the OCL expression in Snippet 2.1 to capture our wanted age constraint.

Snippet 2.1: Constraining the age of a person

```
Context Person inv lifespan:
self.age > 0 and self.age < 130
```

This is an example of an invariant, as can be seen by the use of the inv keyword. Invariants are simply used to specify something that must be true for all instances of a type [17]. For our example we can see that the constraint must be true for all Person objects, by the use of the $Context$ keyword. It is also worth noting that the use of the $self$ keyword, refers to one instance within a context. Although OCL offers other uses and language constructs as well, we will only concern ourself with invariants.

The syntax in Snippet 2.1 is simple and readable, but not completely aligned with our use case. In the end we want to perform transformations on the Abstract Syntax Tree (AST) representations of OCL constraints. To be able to generate these ASTs, the following rewrite rule must first be applied [18]:

> **Context** A inv: Body $\Rightarrow$ A.allInstances $\rightarrow$ forAll(v | Body')
>
> *Where Body' is obtained by replacing all occurrences of self in Body with v.*

Applying this rule on Snippet 2.1 we get the semantically equivalent constraint in Snippet 2.2. Most of our constraints will be written this way from now on, we will however also continue the use of the standard syntax where this makes more sense.

Snippet 2.2: Constraining the age of a person - rewritten

```
Person.allInstances() -> forAll(v | v.age > 0 and v.age < 130)
```

To write the different invariants, the OCL language provides us with a set of basic types, and their belonging operations, some of which we have already touched upon. In addition, all classifiers (classes etc) in the UML model that our OCL expressions are attached to, are

[4]The longest documented lifespan ever recorded is 122 according to wikipedia

also available as types in our OCL expressions [17]. Finally, navigation over associations are available through the common 'dot' notation.

In the next section we will go in more depth on our available language constructs. Note that this is not meant to be a complete overview. Only components that are relevant to our experiments and example transformation will be included. To illustrate the semantics of each operation we will turn to the SimPL model in Figure 2.5.

2.2.2 OCL operations

In this section we will look at the semantics of navigation, and some collection operations, in OCL. Understanding these are enough to later understand the OCL constraints we are using in our experiment. From our experience with working in the ICS context with our user-partner (FMC), these OCL operations have been some of the most commonly used.

2.2.2.1 Navigation

In essence, navigation enables us to start from a specific object, and *"navigate an association on the class diagram to refer to other objects and their properties"*[17]. This is done in OCL using the common dot-notation. For example:

```
objectA.objectB_association
```

Doing this will either result in a collection of 'objectB' objects, or a single 'objectB' object, depending on the multiplicity of the association. If the multiplicity is '0..1' or '1', it results in a single object, otherwise in a collection of objects (which *could* be a collection with only a single object)[17].

2.2.2.2 Collections

Collections in OCL can be of four different types, they each are instances where duplicates[5] are allowed or not-allowed, and where the collection is ordered or unordered. A table enumerating these types can be found in Table 2.1. As a side note, Prolog is an *untyped* language, the only "collection" type that exists is the 'list'. A list can contain duplicates, no duplicates, be ordered, or unordered, it does not matter. Hence, special care should be taken when mapping the different OCL collection types into Prolog, making sure the semantics are preserved.

Collection type	Duplicates allowed	Ordered	Example
Set	no	no	{1,6,2,8}
OrderedSet	no	yes	{1,2,6,8}
Bag	yes	no	{1,6,1,2,8,2}
Sequence	yes	yes	{1,1,2,2,6,8}

Table 2.1: The different collection types in OCL, and their attributes.

Next we will discuss the four collection operations that are utilized in this thesis: Select, Collect, ForAll and IncludesAll.

[5]The same element several times.

Select The select operation enables the extraction of a subset of a collection, based on the result of a boolean expression on each element in that collection. The basic syntax can be seen in Snippet 2.3.

Snippet 2.3: Select

```
Collection -> select(boolean-expression)
```

Example: Say we had a collection of ElectronicConnection objects, but wanted to formulate a constraint around only those that had an ebIndex of 0. Snippet 2.4 shows how this could be written, we could now use the resulting collection to express our wanted constraint.

Snippet 2.4: Select example

```
ElectronicConnection.allInstances() -> select(e | e.ebIndex = 0) -> ...
```

Collect The collect operation enables the creation of a new collection based on evaluating an expression on each element of the original collection. The basic syntax is shown in Snippet 2.5.

Snippet 2.5: Collect

```
Collection -> collect(expression)
```

Example: Using collect we could create a collection of all the SEM objects that are associated with an ElectronicConnection object.

Snippet 2.6: Collect example

```
ElectronicConnection.allInstances() -> collect(e | e.sem) -> ...
```

As noted in [17], the collection type may change when performing a collect operation. See Table 2.2. This will happen because many of the elements in the source collection may have the same extracted value. In Figure 2.5 several ElectronicConnection objects may have the same associated SEM object. Hence applying the collect operation in Snippet 2.6 may give us a collection with multiple copies of the same SEM.

Source	Result
Set	Bag
Sequence	Sequence
OrderedSet	Sequence

Table 2.2: The type of the resulting collection in a select operation, depends on the source collection-type

ForAll The forAll operation, enables the expression of a constraint over all elements in a collection [17]. The basic syntax is shown in Snippet 2.7.

Snippet 2.7: ForAll

```
Collection -> forAll(boolean-expression)
```

Example: Say we wanted to constraint the pinIndex of all ElectronicConnections to be larger than 0. This could be expressed as in Snippet 2.8

Snippet 2.8: ForAll example

```
ElectronicConnection.allInstances() -> forAll(e | e.pinIndex > 0)
```

IncludesAll Similar to forAll, includesAll can also be used to express a constraint. IncludesAll can be used to check if a *source* collection, contains all the elements in a target collection, expressed in the body of includesAll. The syntax is found in Snippet 2.9.

Snippet 2.9: IncludesAll

```
Collection_A -> includesAll(Collection_B)
```

That is, does Collection_A, have all the elements that exist in Collection_B?

Example: Say we wanted to make sure that every SEM object is connected with at least one ElectronicConnection. This could be expressed as in Snippet 2.10.

Snippet 2.10: IncludesAll example

```
ElectronicConnection.allInstances() -> collect(e | e.sem) -> includesAll(SEM.allInstanes())
```

2.2.3 Parsing OCL with Eye OCL Software (EOS)

For our context we only needed a simple and fast way of prototyping a SimPL-model (see Figure 2.5) with attached constraints. EOS [16] aligned well with this use-case, although we found some unfortunate shortcomings in their API that will be discussed next. EOS is a Java library, in the time of this writing, EOS-0.4 is the most recent.

Given that we have an instantiation of the main IEOS class, the code in Snippet 2.11 will give us a (simplified) representation of Figure 2.5.

Snippet 2.11: Creating the SimPL-model with EOS

```
ieos.createClassDiagram();

ieos.insertEnumeration("ElecBoard", new String[]{"8_PIN","16_PIN","32_PIN","64_PIN"});

//Class Electronic Connection
ieos.insertClass("ElectronicConnection");
ieos.insertAttribute("ElectronicConnection", "ebIndex", "Integer");
ieos.insertAttribute("ElectronicConnection", "pinIndex", "Integer");

//Class SEM
ieos.insertClass("SEM");
ieos.insertAttribute("SEM", "eBoards", "ElecBoard"); //not a list!

//Associations
ieos.insertAssociation("ElectronicConnection", "ec","1..*","0..1", "sem", "SEM");

ieos.closeClassDiagram();
```

In our model we wanted the eBoards attribute of class SEM, to be a list of ElecBoard literals. Normally you would use a notations such as $eBoards : ElecBoard[*]$ to achieve this. And so we thought $ieos.insertAttribute("SEM","eBoards","ElecBoard[*]")$ would do the trick.

However we found this notation not to be supported, and did not find an alternative way of achieving the same semantic. This put some limitations on which constraints we could attach to this representation of our model. However we still found it useful enough to continue its use.

Moving on, with the model in place we can now parse constraints attached to the model. The parsing produces a tree representation of an OCL constraint.

Example:

```
ElectronicConnection.allInstances() -> select(e | e.ebIndex > 0) -> size() > 0
```

That is, give us the collection of ElectronicConnections, whose ebIndex is larger then 0. Moreover, make sure that the resulting collection is larger then 0. Parsing this with EOS and our model we get the following:

```
[>(Boolean)[size(Integer)[iterator(select,null)[allInstances(Set(ElectronicConnection))
[constant(0,-1,OclType)]],[>(Boolean,Set(ElectronicConnection))[attribute(ebIndex,Integer)
[variable(0,-1,ElectronicConnection)]],[constant(0,0,Integer)]]]],[constant(0,0,Integer)]]
```

This is not easily readable. To better illustrate these trees we have implemented a transformation to the **dot** format available in the graph visualization tool Graphviz [19]. After the transformation has been performed we can generate the tree found in Figure 2.2.

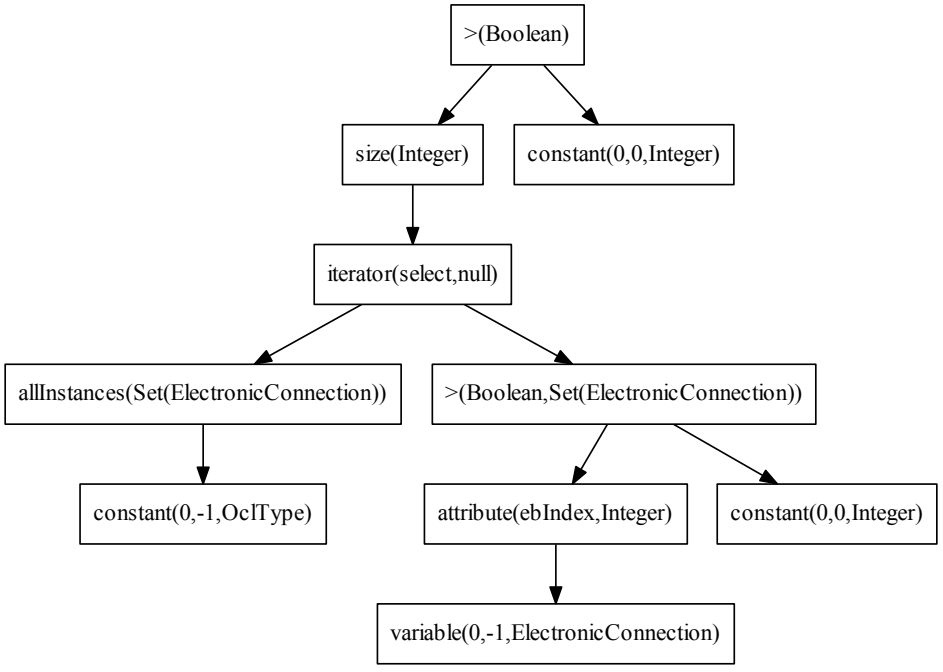

Figure 2.2: Graphical representation of our example constraint

How one should read such a tree is excellently explained in [9]:

> The root of the tree is the most external operation of the OCL expression. The left child of a node is the source of the node (the part of the OCL expression previous to the node). The right child of a node is the body of an iterator expression if the node represents one of the predefined iterators defined in the OCL standard (forAll,

select, etc) or the argument of the operation if the node represents a binary operation (such as '>', union, '+', etc). In this latter case, the source can be regarded as the first operand of the operation.

2.2.4 Constraint complexity

In our context it would be interesting to see if any inherent complexity in our OCL constraints could be predictors of PET. In the event that we were able to consistently do so, one could think of use-cases such as providing warning/feedback to system designers. This is however out of our scope. Our goal is rather to try to uncover if such a relationship exists.

As a starting point we have chosen to adapt the work depicted in [9]. Here, the complexity of an OCL expression is defined as:

> the number of objects that must be considered (i.e. accessed) to evaluate the expression.

Before we move on, we must clarify the difference between the complexity of an OCL expression and the complexity of an OCL constraint. We will adhere to the definition given in [9]. An OCL expression, is the body of a constraint. For example:

```
Context Type
*body/expression*
```

So in the following simple example, "self.ebindex > 0" is the OCL expression. Note that the keyword "Context" is used to specify which class/type we are currently constraining.

Snippet 2.12: Simple example of an OCL constraint

```
Context ElectronicConnection inv:
self.ebindex > 0
```

The implication of this is that one can view the complexity of an OCL expression to only consider *one* object of the context class. On the other hand, the complexity of an OCL constraint will be the complexity of evaluating the OCL expression over *all* objects of the context class[9]. But to start simple, we will start our complexity calculation by only considering OCL expression complexity. What would the complexity in Snippet 2.12 be if we view it as just considering one ElectronicConnection? In this case, the complexity will simply be 1, as there is only one object accessed, namely the 'self' object.

However, the complexity calculation rarely is as straightforward. Consider Snippet 2.13, where we navigate to an associated SEM object.

Snippet 2.13: What is the complexity?

```
Context ElectronicConnection inv:
self.sem -> size() > 0
```

The complexity here depends on two details:

I Multiplicity of the association

II How associations are resolved in the underlying implementation language.

For the purposes of this thesis, we will only consider cases where the multiplicity is 1. This is of course not optimal, and should be addressed in later work. How associations are resolved however, are addressed in subsection 4.2.2, where we discuss our transformation. For the remainder of this section, complexity calculation will be based on the 'contained' method of association resolving (see subsubsection 4.2.2.1). In short, this means that we only have to consider one object when navigating an association, provided that the multiplicity is 1 of course. So with these assumptions in place, what is the complexity in Snippet 2.13? We still have the self object, and since the multiplicity is 1, and we are using containment, only the object of the association is considered in addition. Thus making the complexity equal 2.

We will now turn away from the complexity of only OCL expressions, and will now consider the complexity of the full OCL constraint. To do this, we will use the same rewrite rule as discussed earlier. Lets apply this to the constraint in Snippet 2.13. We then get the following.

Snippet 2.14: Encompassing all objects

```
ElectronicConnection.allInstances() -> forAll(e | e.sem -> size() > 0)
```

To calculate the complexity in Snippet 2.14, we have to introduce some new syntax. N_x will represent the average number of associations to a certain object-x for a collection of objects[9]. Lets say we have 3 ElectronicConnection objects with 3,8 and 16 associated SEM objects respectively. N_{sem} would then equal $\frac{3+8+16}{3} = 9$. Further, we will use P_x to represent the size of the full population of a certain class. So in the case of our example, the number of objects in ElectronicConnection.allInstances() can be represented as $P_{electronicconnection}$, which with our instantiation would equal to 3. To get the complexity of Snippet 2.14, we would then have the following function: $P_{electronicconnection} + P_{electronicconnection} * N_{sem} = 3+3*9 = 30$, which is the correct number of objects accessed to evaluate the OCL constrain. But as mentioned earlier, we have the simplification that associations only will have multiplicity equal to 1. So in our case, the 3 ElectronicConnections would have 1 associated SEM object each. In this case, we don't have to use the N_x notation anymore, since we know that the number of SEM objects accessed is equal to the number of ElectronicConnections objects. For the constraint in Snippet 2.14, we would then simply have the function: $P_{electronicconnection} + P_{electronicconnection} = 3 + 3 = 6$.

Until now we have been manually calculating complexity functions, in the end however we want these functions to be automatically generated for any given constraint. To achieve this we have written several Java implementations of the algorithm found in [9]. We need more than one implementation as we are experimenting with several Prolog representations of the Tool-instance. The different representations requires slightly different OCL operations implementations in Prolog. Navigation over associations have been found to have an especially high impact on complexity. A discussion of the different representations can be found in subsection 4.2.1, and our implementations of the complexity algorithm can be found in Appendix A.

Before we move on, lets have a look at the result of applying this algorithm on the constraint in Snippet 2.14, assuming multiplicity of 1, and 'contained' associations. In Figure 2.3, you will find the AST for this constraint. In each node, "Compl" shows the calculated complexity up to this node (the complexity of its subtrees) [9]. Hence the root node shows the complexity of the whole constraint.

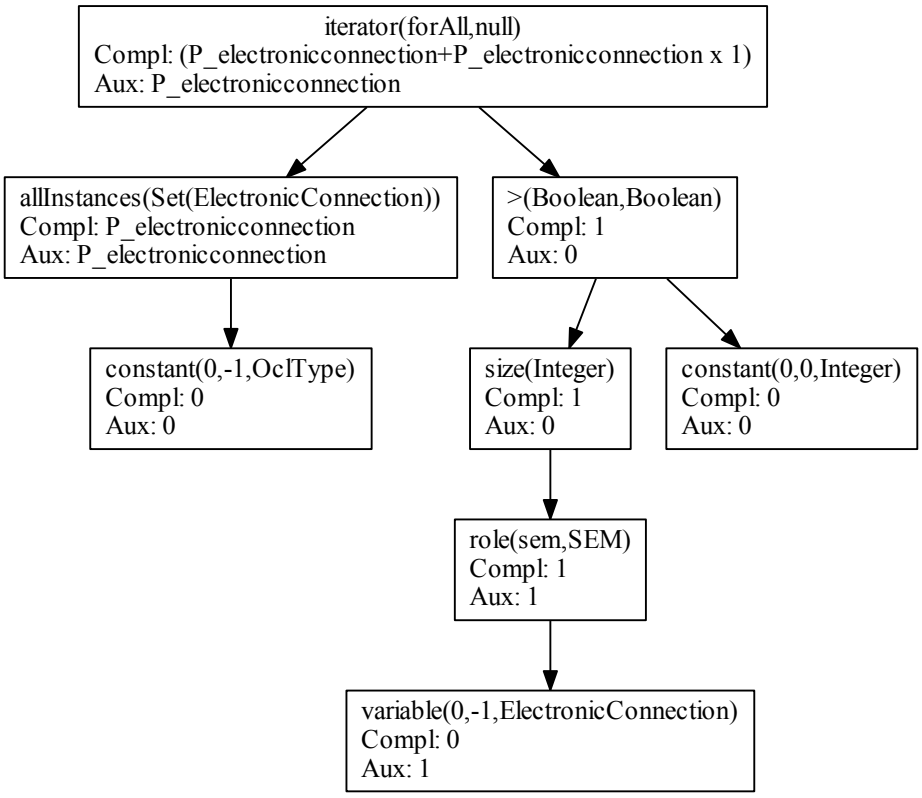

Figure 2.3: Complexity calculation for the constraint in Snippet 2.14 with sem association of 1. And containment for association resolving.

2.3 The SimPL UML-profile

A UML-profile makes it possible to customize UML for a certain need. Amongst others, they are used to add new semantic meaning to *metaclasses* in the UML meta-model. A metaclass is a class, where the instances are also classes[13] (i.e., not objects). So there are metaclasses for the classes: 'package','association','class' and so on, which describe the allowed behavior of these classes. New semantic meaning can be given to these metaclasses by defining *sterotypes*. In Figure 2.4 you will find a small excerpt of the SimPL meta-model, which shows some metaclasses being extended, defining new stereotypes. The system designer can thus both instantiate the metaclasses, and the stereotyped version. Moreover, a UML metaclass can be extended by more than one stereotype. For example, in Figure 2.4, the metaclass 'Dependency' is extended by two stereotypes, namely 'Inherit' and 'RelatedConfigUnit'. Note that each of these stereotypes introduces a distinct concept.

Much of the purpose behind the SimPL profiles' stereotypes is enabling the designer to mark the parts of the model that are configurable. For a detailed explanation of the SimPL UML-profile see [4].

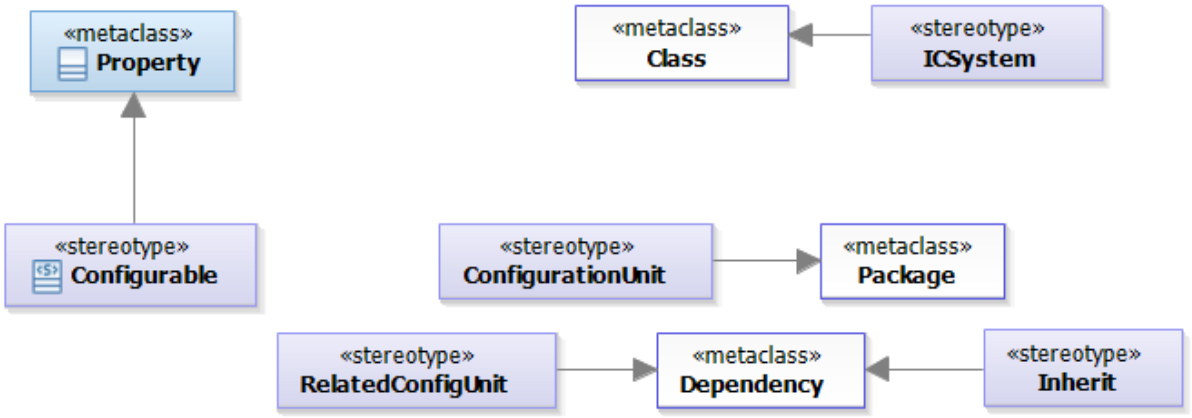

Figure 2.4: An excerpt of the SimPL meta-model

2.3.1 A SimPL model

In this section we will present a sample SimPL-model. This model will lay the foundation for our experiment later, going through the transformation flow presented in Figure 1.2. The model can be found in Figure 2.5, we will now in turn go trough the different elements of the model. Note that it might also help to look at the definitions and mapping rules in section 3.1, for a better understanding. All stereotypes in the model are from the SimPL profile, with the exception of «enumeration», of course. We will not go into detail on the semantic meaning of the naming of elements here. The names comes from an example model, created for our industry partner, FMC.

Packages: There are three packages in the model: *ECConfigurationUnit*, *SEMConfigurationUnit*, and *FMCConfigurationUnit*, all stereotyped as «ConfigurationUnit». Configuration units are connected to classes through a dependency link, stereotyped by «RelatedConfigUnit». The purpose of configuration units is to specify that a certain class is configurable, that is, the end-user can configure the attributes of this class in the configuration process. Configurable attributes are specified through template parameters, visualized as a box on top of the packages. For example, in the *ECConfigurationUnit* (EC is short for ElectronicConnection), we see that the attributes: *ebIndex*, *pinIndex* and *sEM*, are configurable. One of the configurable classes, namely *FMCSystem*, is special. We discuss this next

Classes: There are three classes in the model: *ElectronicConnection*, *SEM*, and *FMCSystem*. A class stereotyped as «ICSSystem» plays a special role in the model. It can be seen as the root/toplevel element, and only one class is allowed to have this stereotype. The configuration process starts from this element, as all other elements can be reached from here. The classes *ElectronicConnection* and *SEM* are configurable, as dictated by their respective configuration units[6].

FMCSystem: Root element, containing $1..*$ ElectronicConnections, and $1..*$ SEMs.

[6]These classes represents special concepts in the subsea oil production system of FMC.

ElectronicConnection: Has two integer attributes, and an association to a SEM object, with multiplicity 0..1. More than one ElectronicConnection can have an association to the same SEM object.

SEM: Has one attribute, an unbound list of ElecBoard literals.

In many of the coming sections. we will refer back to this model to illustrate various points, especially relating to the creation of OCL constraints. In section 3.3, we will transform Figure 2.5 to a Tool-model.

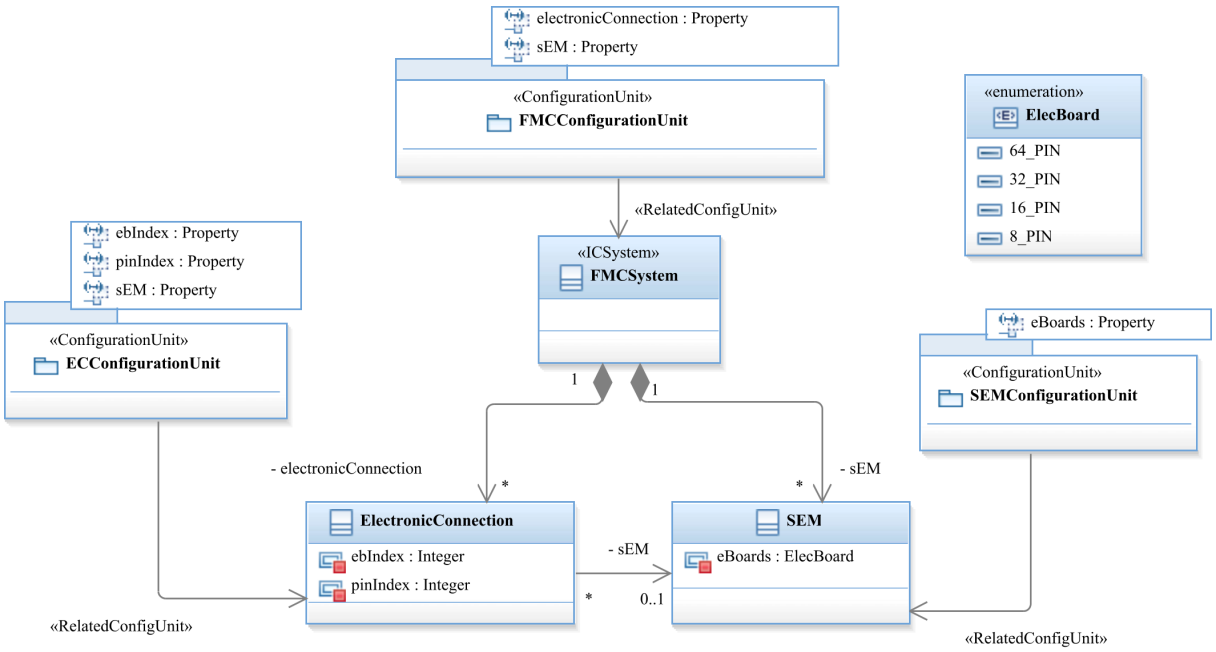

Figure 2.5: A SimPL model

2.4 Solving Constraint Satisfaction Problems with clpfd

As we have earlier noted, finding the valid domains (legal instantiations of attributes) in the Tool-instance, constrained by the Tool-model and OCL constraints, can be seen, and represented as, a Constraint Satisfaction Problem (CSP). This is indeed why we are doing our final transformations to SICStus Prolog, where the models and OCL can be represented as such. Specifically, the CSP can then be solved using the module: 'Constraint Logic Programming over Finite Domains' (clpfd)[11]. Many other constraint programming solutions also exist, a full overview is available at [20]. All of these (we presume), solves CSPs through a technique called *constraint propagation*. Constraint propagation can be defined as *"reasoning which consists in explicitly forbidding values or combinations of values for some variables of a problem because a given subset of its constraints cannot be satisfied otherwise"*[21]. That is, with constraint propagation, one is actively pruning/reducing the valid domains of variables through applying constraints. Hence, one can immediately see if the application of a constraint would

invalidate the domain of some variable (making the valid domain empty). CSPs, using constraint logic programming is elaborately discussed in [22]. Here, any given *computation state* in a CSP, is described as having two parts:

I The *goal part: "the conjunction of goals that remains to be solved"*.

II The *constraint store: "the set of constraints accumulated up to this point of execution"* (the result of constraint propagation).

These parts can then be represented as $\langle\, G \,\square\, \sigma \,\rangle$, where G is the goal part, and σ is the constraint store. Furthermore, ϵ is used if any of the two parts are empty [22].

Example:

Say we had the Prolog code in Snippet 2.15. Note that the clpfd module must first be loaded, comments are given after $\%$. We start the execution by calling $entry(X, Y)$.

Snippet 2.15: A simple CSP

```
:- use_module(library(clpfd)).

entry(X,Y):-
        domain([X,Y], 0,10), % prune the domain of X and Y to be between 0 and 10
        Y #< X,              % constrain Y to be less than X
        next(X,Y).

next(X,Y):-
        X #= Y*2.            % X is constrained to be Y times 2
```

Below is the computation steps taken to solve the CSP. On the left hand side we have the current state, and on the right hand side the current valid domains of X and Y given the current constraint store. Note that inf represents $-\infty$, and sup represents $+\infty$ in clpfd.

$$
\begin{aligned}
\langle\, entry(X,Y) \,\square\, \epsilon \,\rangle &\qquad X \in inf..sup, Y \in inf..sup \\
\langle\, domain([X,Y],0,10) \,\square\, \epsilon \,\rangle &\qquad X \in inf..sup, Y \in inf..sup \\
\langle\, Y\# < X \,\square\, domain([X,Y],0,10) \,\rangle &\qquad X \in 0..10, Y \in 0..10 \\
\langle\, next(X,Y) \,\square\, domain([X,Y],0,10), Y\# < X \,\rangle &\qquad X \in 1..10, Y \in 0..9 \\
\langle\, X\# = Y*2 \,\square\, domain([X,Y],0,10), Y\# < X \,\rangle &\qquad X \in 1..10, Y \in 0..9 \\
\langle\, \epsilon \,\square\, domain([X,Y],0,10), Y\# < X, X\# = Y*2 \,\rangle &\qquad X \in 2..10, Y \in 1..5
\end{aligned}
$$

At this point we have reached one of the terminal states, which are either [22]:

I The goal part is empty (ϵ). That is, a solution to the CSP has been found, which is our case.

II No clause can be applied to the current goal. That is, a solution can not be found without possibly backtracking.

III The current goal/constraint is not satisfiable with the current constraint store. This would happen if we called $entry(X, 10)$ for example. Here we would fail at line 3, 10 cannot be less than any of the valid instantiations of X (0..10).

Prolog would then answer:

```
X in 2..10,
Y in 1..5 ?
yes
```

Notice that Prolog here outputs the full valid domains of X and Y. Not all values in X's domain are valid over all the values in Y's domain. It can rather be read as; given that you choose a value between 2 and 10 for X, I will promise that there exists a value between 1 and 5 in Y, so that the instantiations still satisfy the constraints.

More often than not, people solving CSPs are interested in an instantiation of their variables, rather than the valid ranges. Maybe they were interested in maximizing the instantiation of their variables, maybe minimizing? There is a special predicate for doing this in clpfd, namely $labeling/2$. This predicate performs a search, trying to make all (domain) variables *ground*, which is the technical word for saying that a variables' domain only contain one value. How the search is carried out can be controlled by providing options to the predicate, and one can indeed also specify if one wants to maximize the domain of a variable. Lets say we wanted to maximize X, we would then call:

```
| ?- entry(X,Y),labeling([maximize(X)],[X]).
```

giving us:

```
X = 10,
Y = 5 ?
yes
```

In our context however, aiding end-users in a configuration process, this is not the desired behavior. We want to inform the end-user of the valid domains of variables, rather than choosing the values for them. Still, there might exist some cases where a value should be maximized or minimized at some point in the configuration process. Moreover, there might be cases where we want better control over the backtracking when searching for a solution. We have not encountered such cases yet, but it is good to keep in the back of ones mind, that labeling then could be utilized.

Chapter 3

First transformation step: SimPL2Tool

The configuration tool presented in [7], does not handle SimPL-models directly. Rather, it has it's own internal representation of the product line. In this representation, only the configurable elements (see section 1.1) of the original model are contained. Hence it is optimized for purposes of the configuration process. The meta-model for this internal representation is given in its full in Figure 3.1, this meta-model will be referred to as the *Tool-metamodel*, or *Tool* for short. This chapter will go into details on how the SimPL to Tool transformation was implemented. The complete set of mapping rules are given in section 3.1, and an excerpt of our implementation is found in section 3.2.

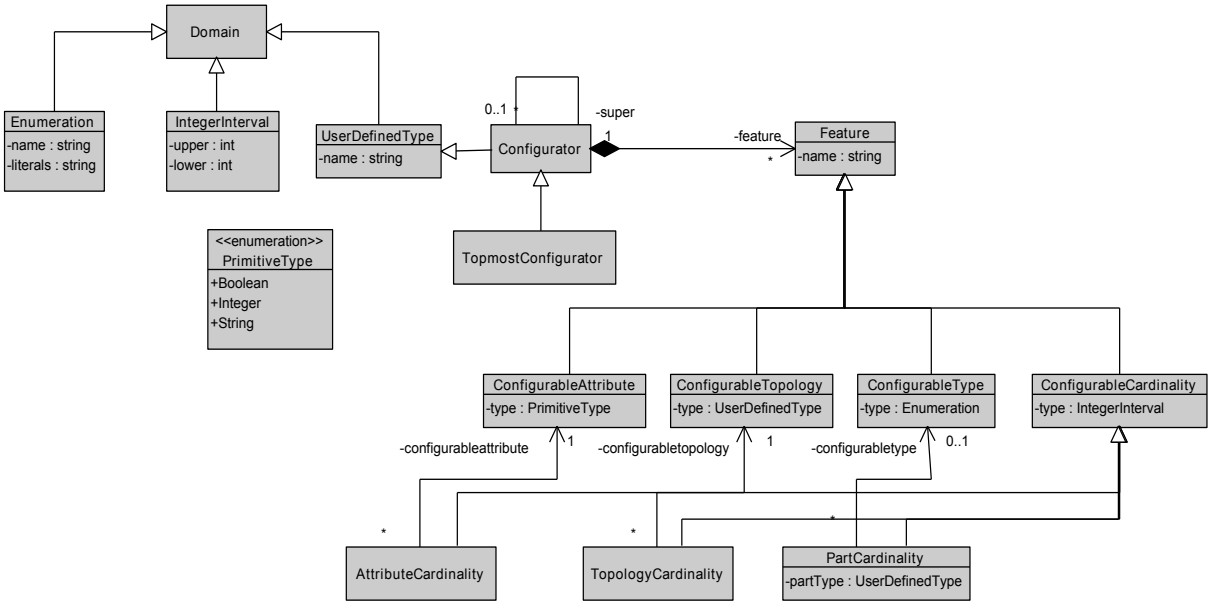

Figure 3.1: The Tool-metamodel

Before looking at the mapping rules, a brief explanation of the Tool-metamodel is in place. The main focus of attention should be on the *Configurator* class. Configurators contain a set of *features*, which for example can be configurable attributes, like an integer. It can also be a configurable type, which means that there is a choice in implementation class. A configurable topology, means there exist associations to other classes from this class, and one is interested

in configuring the multiplicity of the association end. Finally, the configurable cardinalities, contains information about the multiplicities of the previously mentioned features.

3.1 Mapping rules

To define the mapping rules of the transformation, some definitions are first needed. These are mainly used to make it clear which elements in the SimPL-model we are referring to. The definitions are as follows:

Definitions

Configurable Class: A class that is associated with a configuration unit (a template package stereotyped by «ConfigurationUnit») is called configurable.

Configurable attribute: An attribute of a configurable class is called configurable if, and only if, it is pointed by a template parameter of the corresponding configuration unit.

Configurable association-end: In UML, each association-end is owned by a class (i.e., the class at the other end of the association) and is usually accompanied by a role-name. Such an association-end represents a property of the owner class. An association-end is called configurable if, and only if, it is pointed by a template parameter of the configuration unit of the owner class.

Configurable part: In a whole-part relationship (implemented by a composite association relation in UML), if the class on the whole side is configurable, and the part is pointed by a template parameter, then the part (denoted by its rolename) represents a configurable part. We refer to the class at the part side of the composition association as the type of the configurable part.

SuperClass: In the following mapping rules, we use "SuperClass" to refer to a class that is at the root of a generalization hierarchy.

With the definitions in place, the mapping rules can now be defined. They are as follows:

Mapping rules

Configurable Class-to-Configurator: Every configurable class in SimPL is mapped to a Configurator in Tool with the same name.

A: If the configurable class (sub-configurable-class) is related to another configurable class (super-configurable-class) via an «Inherit» dependency, then in the target model, the Configurator representing the super-configurable-class becomes the target of the attribute "super" in the Configurator representing the sub-configurable-class. (Note that the «Inherit» dependency connects the configuration units of the configurable classes.)

B: Configurable ICSystem-to-TopmostConfigurator: A configurable class stereotyped by «ICSystem» is mapped to a TopmostConfigurator

Configurable Attribute-to-ConfigurableAttribute: Every configurable attribute (SimPL) of a class is mapped to a ConfigurableAttribute in Tool with the same name. The ConfigurableAttribute should be owned by the corresponding Configurator, and should have the same name and the same type as the original attribute.

Configurable Association-end-to-ConfigurableTopology: Every configurable association-end is mapped to a ConfigurableTopology. The name of the ConfigurableTopology is the same as the role-name of the association-end. The type of the ConfigurableTopology is the UserDefinedType denoting the transformation of the Class connected to the association-end.

Generalization hierarchy-to-ConfigurableType: For every configurable part typed by a SuperClass, we add a ConfigurableType to the corresponding configurator. The name of the ConfigurableType is built by concatenating the string "_type" to the end of the name of the configurable part. The type of the ConfigurableType is an Enumeration with literals equal to the subclasses (names) of the type of the configurable part. The name of the Enumeration should be the name of the type of the Configurable part concatenated with "Subtypes".

Multiplicity-to-ConfigurableCardinality: Every unfixed multiplicity is mapped to a ConfigurableCardinality as follows

A: Multiplicity on attributes: for every configurable attribute with unfixed multiplicity (i.e., a multiplicity of the form l..k, where k > l), an AttributeCardinality should be added to the target model. This AttributeCardinality should have a pointer to the ConfigurableAttribute representing the configurable attribute.

B: Multiplicity on association ends: for every association end with unfixed multiplicity, a TopologyCardinality should be added to the target model. This TopologyCardinality should have a pointer to the corresponding ConfigurableTopology.

C: Multiplicity on configurable parts: for every configurable part with an unfixed multiplicity, a PartCardinality should be added to the target model. The name of the PartCardinality is built by concatenating the string "_card" to the end of the name of the configurable part. The partType attribute of the PartCardinality is set to the UserDefinedType denoting the transformation of the type of the configurable part. If the type of the configurable part is a SuperClass, then a pointer to the related ConfigurableType should be added.

3.2 ATL implementation

Before choosing a transformation language, we evaluated some properties that should be supported by the chosen language.

I Product Family Models can be large, up to several thousands elements in each model. So the chosen language should be scalable.

II There are many dependencies among elements in the Tool-metamodel. By dependencies among elements, we mean that initialization of elements cannot happen in an arbitrary order. Some elements need to be instantiated, before other can be instantiated. This should be possible in the chosen language.

III Since our configuration tool potentially will be used in real products, it is important that the transformation code is easily maintainable, and easy to understand.

ATL has been found to be one of the most scalable transformation languages, when compared to other languages [23][24]. Further, postponed initialization of elements are possible through so called lazy-rules [25] (explained later). Finally, ATL is a declarative language, and so is inherently modular, which supports maintainability. Further, it has a lively community [26], and even possibilities of commercial support offered by Obeo, the original developer of ATL [27].

The finished implementation in ATL had the characteristics found in Table 3.1. In ATL there are three main ways of writing mapping rules: matched-rules, lazy-rules and unique-lazy-rules [25]. By declaring matched rules, you simply state which source element should be matched, and how the corresponding element in the target model should be initialized. We only use matched rules in two cases, namely for the two types of *configurable classes*, which are mapped to *Configurators* in Tool. These define our main mapping rules, where we prepare/find the other elements that will be mapped to *Features* in Tool. To find these elements we take advantage of an ATL construct called *helpers*. Helpers can be viewed as the ATL equivalent of methods. In Snippet 3.1 we have an example of one of our defined helpers, this helper operates in the context of a SimPL Class, and returns the collection of primitive attributes of that class. A lot of the preparation for the rest of the transformation is done inside our two matched rules, since all Features are owned/contained by their respective Configurators (see Figure 3.1). Snippet 3.2 shows the main structure of how the configurableClass2Configurator matched rule is implemented in ATL.

	Number of Rules	Approx. Lines of code
Matched Rules	2	250
Unique Lazy Rules	6	100
Helpers	11	100

Table 3.1: Characteristics of the ATL implementation

Snippet 3.1: ATL helper example.

```
helper context SimPL!Class def : getPrimitiveAttributes : SimPL!Property =
  self.getAllAttributes() -> select(e | e.type.oclIsKindOf(SimPL!PrimitiveType));
```

Snippet 3.2: ConfigurableClass to Configurator

```
rule configurableClass2Configurator {
  from configurableClass : SimPL!ConfigurableClass {
    *initialize local variables*
  }
  to  configurator : Tool!Configurator{
    *specify how the target element should be constructed*
    }
  }
```

All other mapping rules are nested inside this rule, and so they are *called* instead of *matched*, for this we use unique-lazy-rules. Unique-lazy-rules can be thought as mapping rules you explicitly call when needed. The *unique* keyword stem from the fact that the same source element never will create more than one target element (as opposed to matched and lazy rules) [25]. The unique-lazy-rule for mapping configurable attributes to *ConfigurableAttributes*, can be found in Snippet 3.3. It gets source elements of type Property (SimPL) as input, and performs a simple mapping to a *ConfigurableAttribute* in Tool. The rule for mapping multiplicity to Attribute Cardinalities can be found in Snippet 3.4. Here we actually have two sources: An already mapped configurable attribute (Tool), and the SimPL property that was used to generate this configurable attribute. Further, we also have two targets: First an Attribute Cardinality, which gets a reference to the input Configurable Attribute, and also a reference to the second target, an Integer Interval. The Integer Interval is initialized using the input SimPL Property.

Snippet 3.3: ConfigurableAttribute to ConfigurableAttribute

```
unique lazy rule configurableAttr2ConfigurableAttr {

  from property : SimPL!Property

  to
  configurableAttr : Tool!ConfigurableAttribute(
    name <- property.name,
    type <- property.type
  )
}
```

Snippet 3.4: Multiplicity to AttributeCardinality

```
unique lazy rule multiplicity2AttributeCardinality {

  from
     configurableAttr : Tool!ConfigurableAttribute,
     property : SimPL!Property

  to
  attrCard : Tool!AttributeCardinality (
    name <- configurableAttr.name + '_cardinality',
    type <- integerInterval, % creates reference to integerInterval
    configurableattribute <- configurableAttr % creates reference to configurableAttr
  ),
  integerInterval  : Tool!IntegerInterval(
    lower <- property.lower,
    upper <- property.upper
```

```
  )
}
```

3.3 Example transformation

In this section we will look at the result of transforming the SimPL-model presented in subsection 2.3.1, and Figure 2.5. The transformed SimPL-model can be found in Snippet 3.5.

Lines 1-5: *FMCSystem* has been mapped to a *TopmostConfigurator* since it was stereotyped by «ICSSystem». Further, is has four features. The first two features is a result of the "Generalization hierarchy-to-ConfigrableType" rule, having references to the two *Configurators*. While the last two features are generated by the "Multiplicity-to-ConfigurableCardinality:C" mapping rule, specifying the numbers of configurators that can be created.

Lines 7-11: ElectronicConnection has been mapped to a Configurator, since a package stereotyped by «ConfigurationUnit», had a dependency relation (stereotyped by «RelatedConfigUnit») to this class. It has four features. The first two are the two ConfigurableAttributes (as pointed by the template parameters on this class' ConfigurationUnit), which are a result of the "Configurable Attribute-to-ConfigurableAttribute" mapping rule. We then have a ConfigurableTopology, representing the configurable association to a SEM object, which is a result of the "Configurable Association-end-to-ConfigurableTopology" mapping rule. Finally, we a TopologyCardinality, specifying the possible multiplicity of the association, which is a result of the "Multiplicity-to-ConfigurableCardinality:B" mapping rule.

Lines 13-18: A mapping of the ElecBoard enumeration. Note that we have cheated here, enumerations are currently not properly handled by the ATL transformation.

Lines 19-22: SEM has been mapped to a Configurator, by the same rules as for ElectronicConnection. It has two features. A ConfigurableTopology, representing the list of ElecBoard literals, and a TopologyCardinality, specifying the multiplicity of this list.

Note that references are specified by */x/@feature.y*, where x is the x'th element in the file, and y is the y'th element within x (counting from zero). We will later see how an instance of the Tool-model can be represented-as/transformed-to a Prolog query in subsection 4.2.1.

Snippet 3.5: A Tool-model

```
1  <Tool_MM:TopmostConfigurator name="FMCSystem">
2    <feature xsi:type="Tool_MM:ConfigurableType" name="electronicConnection_type" type="/1"/>
3    <feature xsi:type="Tool_MM:ConfigurableType" name="sEM_type" type="/3"/>
4    <feature xsi:type="Tool_MM:PartCardinality" name="electronicConnection_type_card" type="/4"
         partType="/0/@feature.0"/>
5    <feature xsi:type="Tool_MM:PartCardinality" name="sEM_type_card" type="/4" partType="/0/
         @feature.1"/>
6  </Tool_MM:TopmostConfigurator>
7  <Tool_MM:Configurator name="ElectronicConnection">
8      <feature xsi:type="Tool_MM:ConfigurableAttribute" name="ebIndex" type="Integer"/>
9      <feature xsi:type="Tool_MM:ConfigurableAttribute" name="pinIndex" type="Integer"/>
10     <feature xsi:type="Tool_MM:ConfigurableTopology" name="sEM" type="/3"/>
```

```
    <feature xsi:type="Tool_MM:TopologyCardinality" name="sEM_cardinality" type="/5"
        configurabletopology="/1/@feature.2"/>
</Tool_MM:Configurator>
<Tool_MM:Enumeration name="eBoardsSubtypes">
  <literals>8_pin</literals>
  <literals>16_pin</literals>
  <literals>32_pin</literals>
  <literals>64_pin</literals>
</Tool_MM:Enumeration>
<Tool_MM:Configurator name="SEM">
    <feature xsi:type="Tool_MM:ConfigurableTopology" name="eBoards" type="/2"/>
    <feature xsi:type="Tool_MM:TopologyCardinality" name="eBoards_cardinality" type="/4"
        configurabletopology="/3/@feature.0"/>
</Tool_MM:Configurator>
<Tool_MM:IntegerInterval upper="-1"/>
<Tool_MM:IntegerInterval upper="1"/>
```

Chapter 4

Second transformation step: UML/OCL2Prolog

As mentioned in chapter 1, the configuration framework requires two steps of transformation. In this section, we discuss the second step of transformation, which is the UML/OCL2Prolog transformation. A major consideration for this transformation is the execution time of the transformed Prolog code. Failure to construct efficient Prolog code is expected to limit user adoption of our configuration framework.

We consider the mappings in our transformation to belong to two categories; simple constructs, where we see one natural mapping to Prolog. And more complicated constructs, for which we have identified several alternative mappings to Prolog. Our main focus will be on the last category. That is, mappings where we suspect choosing one solution over the other will matter in terms of PET. We will finish this chapter by looking at two different example sets of OCL constraints, and the transformation of these. We will later use these constraints in our experiment.

To better exemplify the mapping alternatives, we will refer to the example Tool-instance[1] depicted in Figure 4.1. Note that this is a simplified instance of the Tool-model presented in Snippet 3.5.

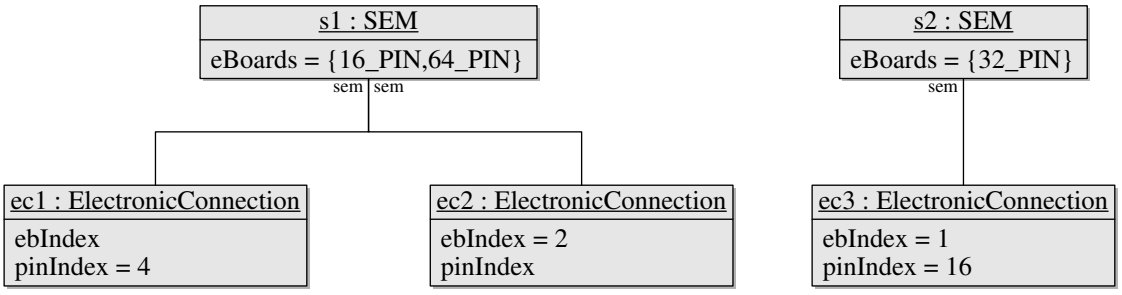

Figure 4.1: The Tool-instance consists of 2 SEM objects and 3 ElectronicConnection objects. Both ec1 and ec2 has an association to s1, while ec3 has an association to s2.

[1] Here we are giving an instance of the Tool-model as a (UML) object-model diagram. Normally these instances are realized in Java

4.1 Three sources, two targets

Before we delve into our different mappings, we should take care to introduce the different sources and targets of our transformation. In Figure 1.2 we attempted to illustrate our situation

The first source is the Tool-model, which we have an example of in Snippet 3.5. A set of OCL constraints is attached to the Tool-model[2]. These constraints are written in the context of classes in the Tool-model, specifying the legal instantiations of the model(see section 4.3 for an example). The Tool-model + OCL constraints must be transformed to a set of Prolog predicates, having the same semantics.

This brings us to the Product under Configuration, or as we shall refer to it, the Tool-instance. The Tool-instance is the real-world-realization of the Tool-model, which currently is done with Java. Now, this demands a clarification. We are including the transformation from a Tool-instance to a Prolog query, as part of the UML/OCL2Prolog transformation. In reality, we are going via Java. One can consider this Java to Prolog transformation as being part of the realization of the UML/OCL2Prolog transformation. In Figure 1.2, the Tool-instance is represented with the Prolog variable "Instance". This Prolog representation of the Tool-instance, will be evaluated over the predicates earlier defined by the Tool-model + OCL.

4.2 Choice points

In this section we will enumerate the mappings where we have found that there exists more than one solution. We do not claim that our solutions in any way are exhaustive. And other, even better solutions might exist. The subset of our transformation that by far will receive the most attention is the following mappings:

Mapping the Tool-instance to Prolog representation :

The Tool-instance will be mapped to a Prolog query. How should this query be structured?

Mapping associations in the Tool-instance to Prolog representation :

How should associations be handled in Prolog? This is really a two part problem. Firstly it must be decided how associations should be represented structurally (the syntax of the query). Secondly, these associations must be navigable.

Mapping our set of OCL constraints to an organized set of Prolog predicates :

When OCL constraints are transformed to Prolog, how should they be organized/structured? That is, does there exist certain organizations of our Prolog predicates that yields lower PET than others? Our starting point for predicate organization is that predicates operating on the same class/context will exist together. We will however go further than this.

[2]These OCL constraints are originally written in the context of configurable classes in the SimPL-model. Since the corresponding *Configurators* in the Tool-model will have the same name, we believe it will be non-problematic, or at least easy, to transfer the OCL from the SimPL-model to the Tool-model.

How the Tool-instance is represented, and how navigations are resolved, has an inherit impact on the implementations of the various OCL operations. Within each following section we will therefore also give details on how a selection of OCL operations were implemented in Prolog. This collection of implementations makes our OCL-in-Prolog API. In the API you will for example find two implementations of the OCL collection operation "Select", where each implementation are designed to work together with the respective Tool-instance representations. The full source code can be found in Appendix B.

4.2.1 Tool-instance representation

We have come up with two different solutions for representing the Tool-instance as a Prolog query. The first one is a pure list of lists structure. The second representation uses associations lists, which is a special form of binary trees.

4.2.1.1 List representation

Given that we have the query *check_system(Tool_instance)* as input to Prolog, *Tool_instance* will take the following form:

```
Tool_instance -> [Class_1,Class_2,...,Class_n]
Class_n -> [Object_1,Object_2,...,Object_m]
Object_m -> [id-c-o,name_1-v_1,name_2-v_2...,name_k-v_k]
```

For the list representation we are assuming that we know the order of the class list in Tool_instance, and the internal order of objects in that list. Moreover, that we also know the order of attributes within an object. Basing of this assumption, note that an objects' first argument is its ID on the following form: id-c-o. Where c is the index of the objects' class list in Tool_instance, and o is the index of this object in that class list.

Example: An object with ID = id-1-2, would exist in the first list in Tool_instance, and would be the second element in that list. This is important for the later discussion of navigations. In Snippet 4.1, this notation has been used on the example Tool-instance. Note that only ec1 and s1 is included in expanded form because of space limitations. Note also that we do not show here how associations are resolved, as this will be discussed in subsection 4.2.2.

Snippet 4.1: Tool-instance in list representation

```
check_system([[id-1-1,ebindex-EbindexID,pinindex-4,s1],ec2,ec3],[[id-2-1,eboards-[16_PIN,64
    _PIN]],s2])
```

As previously discussed in subsection 2.2.2, we are only concerning ourself with a subset of OCL. And in relation to the OCL operations, we will only implement Select, Collect and IncludesAll. For both the Tool-instance representations, these operations require a different prolog implementation to work together. The rest of this section will concern our implementations of these operations, for our list representation.

Implementation of Select For our implementation of Select we have taken advantage of the include/3 predicate available in the 'lists' library of SICStus Prolog. include/3 has the following documentation [28]:

include(:Pred, +Xs, ?SubList)

succeeds when SubList is the sublist of Xs containing all the elements Xi[,Yi[,Zi]] for which Pred(Xi[,Yi[,Zi]]) is true. That is, it retains all the elements satisfying Pred.

This is quite aligned with with our understanding of select (see section 2.2.2.2), and we can treat Pred. as our boolean expression (the body of Select). Our implementation can be found in Snippet 4.2.

Snippet 4.2: Implementation of Select for list representation

```
select_list(NavLeft,Rel,NavRight,List,Tool_instance,NewList):-
       include(check_attribute_list(NavLeft,Rel,NavRight,Tool_instance),List,NewList).
```

Note that our implementation is not generally applicable. In this state we are only able to handle two sets of navigation steps, and applying a binary boolean relation on the two navigation endpoints. By endpoint we are referring to the result of a navigation, be it an attribute or an object.

However, we have also included support for $NavLeft$ and $NavRight$ being integers to perform direct comparison. This is enough to perform our experiment, but a more general implementation will of course be needed later. Here follows an explanation of the arguments of select_list/6:

NavLeft: A list of navigation steps on the form [step_1,step_2,...,step_n]. Or an integer.

Rel: A boolean binary relation such as '>' or '=', between the endpoints of NavLeft and NavRight.

NavRight: As NavLeft

List: The list of objects to iterate over.

Tool_Instance: The full Tool-instance. Needed to resolve navigations when associations are not contained.

NewList: The output list.

As the Pred. argument to include/3 we have implemented the predicate check_attribute_list/5, which performs the navigations and applies the relation to the results of the navigations. The implementation of check_attribute_list/5 can be found in Snippet 4.3.

Snippet 4.3: check_attribute_list/5

```
check_attribute_list(NavLeft,Rel,NavRight,Tool_Instance,List):-
        (integer(NavRight) -> ValueRight = NavRight ; navigate_list(NavRight,Tool_Instance,
            List,ValueRight)),
        (integer(NavLeft) -> ValueLeft = NavLeft ; navigate_list(NavLeft,Tool_Instance,List,
            ValueLeft)),
        call(Rel,ValueLeft,ValueRight),!.
```

We here first perform an Integer check on NavRight/Left to allow direct values, otherwise the navigation is performed. navigate_list/4 will be explained in subsection 4.2.2.

We will now look at a brief example before moving on to Collect. We will use the example from Figure 4.1.

Example: Lets say we had a collection of the ElectronicConnections objects and wanted a collection of those that had an ebIndex of 1. The OCL would look like:

```
ElectronicConnection.allInstances() -> select(e | e.ebindex = 0) ...
```

And the corresponding call to select_list/6 in Prolog would then look like:

```
select_list([ebindex],#=,1,ElectronicConnections,Tool_Instance,NewList)
```

And the resulting NewList would be unified as:

```
NewList = [ec1,ec3]
```

ec3 is included since its ebIndex was already configured to be 1, and ec1 was included as its ebIndex was currently unconfigured and could be set/constrained to be 1.

Implementation of Collect For our implementation of Collect we will take advantage of maplist/3 which is included in the 'lists' library of SICStus Prolog. It has the following documentation [28]:

> maplist(:Pred, +OldList, ?NewList)
>
> *succeeds when Pred(Old,New) succeeds for each corresponding Old in OldList, New in NewList. Either OldList or NewList should be a proper list.*

So maplist/3 iterates OldList, and for each element, apply $Pred$ as a function from this element to the New element. You can consider this as a mapping from the element in OldList to the new element in NewList. As $Pred$ we will use our navigation predicate for lists, navigate_list/4. Our implementation for collect_list/4 can be found next.

Snippet 4.4: Implementation of Collect for list representation

```
collect_list(NavList,List,Tool_Instance,Result):-
       maplist(navigate_list(NavigationsList,Tool_Instance),List,Result),!.
```

The arguments are as follow:

NavList: List of navigation steps as for NavLeft/Right in select_list/6.

List: The list of object to iterate over.

Tool_Instance The full Tool-instance. Needed to resolve navigations when associations are not contained.

Result: The output list, with the collected elements.

Example: Let say we had a collection of SEM objects, and wanted a collection of their eBoards. The OCL would be:

```
SEM.allInstances() -> collect(e | e.eBoards) ...
```

Which gives us the following call to collect_list/4:

```
collect_list([eboards],SEMs,Tool_Instance,Result)
```

And Result would be:

```
Result = [[16_PIN,64_PIN],[32_PIN]]
```

Which is simply the list of eBoards of our SEM objects.

Implementation of IncludesAll For our implementation of includes_all_list/2, we will take advantage of two predicates, the built in sort/2 and subseq0/2 from the 'lists' SICStus Prolog library. subseq0/2 has the following documentation [28]:

> subseq0(+Sequence, ?SubSequence)
>
> *is true when SubSequence is a subsequence of Sequence, but may be Sequence itself. Thus subseq0([a,b], [a,b]) is true as well as subseq0([a,b], [a]). Sequence must be a proper list, since there are infinitely many lists with a given SubSequence.*

We almost have the functionality we need with subseq0/2, but the following case will explain why we also need sort/2. Lets say we have list A = [a,b,d,c] and list B = [a,d,c,b,e,f], and want to know if all the elements in A also exist in B. Which is the semantics of IncludesAll. Calling subseq0(B,A) directly would fail, as it tries to find the full sequence [a,b,d,c] in B. However, these elements does not exist in B as a coherent sequence. Only sequences like [a,d,c] or [c,b,e,f] would work in this case. Because of this we must sort the two lists first, giving us A = [a,b,c,d] and B = [a,b,c,d,e,f]. The documentation for sort/2 can be found next [11]:

> sort(+List1, -List2)
>
> *Sorts the elements of the list List1 into the ascending standard order, and removes any multiple occurrences of an element. The resulting sorted list is unified with the list List2.*

Now A exist as a sequence in B, and subseq0(B,A) would succeed. Our implementation can be found in Snippet 4.5.

Snippet 4.5: Implementation of IncludesAll for list representation.

```
includes_all(_,[]):-!.
includes_all(Whole,Part):-
        sort(Part,PartSorted),
        sort(Whole,WholeSorted),
        subseq0(WholeSorted,PartSorted),!.
```

The arguments are as follows:

Whole: The list to check against.

Part: The list to check if all its elements exist in $Whole$. Must be equal to or smaller in size than Whole.

Note that we first check if Part is the empty list, in which case we can succeed at once.

4.2.1.2 Representation as association lists

Association lists (assoc) in SICStus Prolog Prolog are implemented as AVL trees. Firstly this means that they are an example of binary search trees, and secondly that they are subject to the Adelson-Velskii-Landis balance criterion [28].

Binary search trees conform to the following criterion [29]:

> A binary tree where every nodes' left subtree has keys less than the node's key, and every right subtree has keys greater than the node's key.

AVL trees augments this with the following criterion [30]:

> A balanced binary search tree where the height of the two subtrees (children) of a node differs by at most one. Look-up, insertion, and deletion are O(log n), where n is the number of nodes in the tree

Often AVL tree implementations are also self balancing, meaning that the insertion/deletion of nodes, will automatically cause a *rebalancing* of the updated tree such that it still conforms to the criteria above. This is *not* the case for SICStus Prolog assoc implementation [28]. However, we did not find this to cause any problems. But this might be related to us only dealing with a subset of OCL, and a migration to a self balancing implementation might be needed/preferred if one want to transform the full OCL.

A node in an assoc takes the following form:

```
assoc(Key,Value,LeftSubtree,RightSubtree)
```

And leaf nodes are just the atom *assoc*. An assoc with just one node hence looks like:

```
assoc(Key,Value,assoc,assoc)
```

With this in mind, we will next look at the solution for representing the Tool-instance as assoc. Given that we have the query *check_system(Tool_Instance)*, Tool_Instance will take the following form:

```
Tool_Instance -> assoc(ClassKey,ObjectsAssoc,LeftSubtree,RightSubtree)
ObjectAssoc -> assoc(Object_ID,AttributeAssoc,LeftSubtree,RightSubtree)
AttributeAssoc -> assoc(AttributeKey,Value,LeftSubtree,RightSubtree)
```

In Figure 4.2 we have attempted to visualize this structure.

The main advantage that our assoc representation have over our list representation, is logarithmic access to any given node/element in each collection. This is done using the predicate get_assoc/3 in the 'assoc' library. It has the following documentation [28]:

> get_assoc(+Key, +Assoc, -Value)
>
> *assumes that Assoc is a proper "assoc" tree. It is true when Key is identical to (==) one of the keys in Assoc, and Value unifies with the associated value. Note that since we use the term ordering to identify keys, we obtain logarithmic access, at the price that it is not enough for the Key to unify with a key in Assoc, it must be identical. This predicate is determinate. [...]*

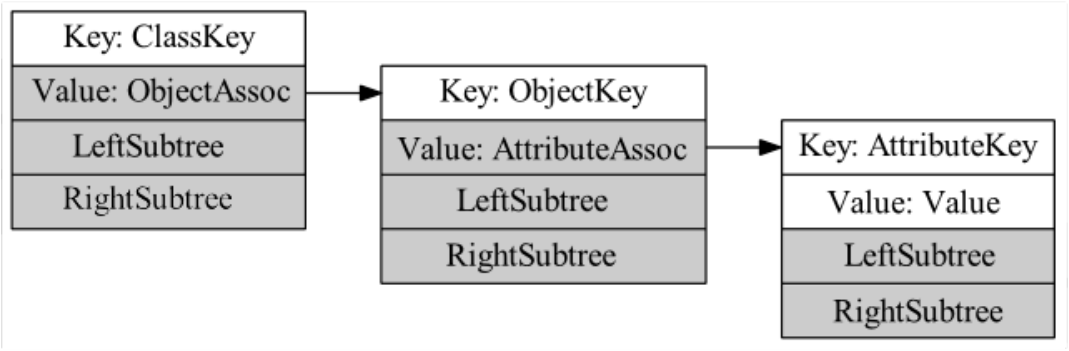

Figure 4.2: Visualization of our assoc structure. Consider each compartment as an argument in the SICStus Prolog assoc node implementation. The three boxes are assoc, as well as the gray compartments.

So to exemplify, Assume that we have objects of C different classes in the Tool-instance, and there are O number of objects of the class we wanted, and the object had A attributes. We could traverse from the most outer assoc to any given attribute with cost O(log(C) + log(O) + log(A)). The same big-Oh for lists would be O(C + O + A).

In Snippet 4.6 you will find the assoc representation of the Tool-instance running example. Note that ec1,ec3 and s2 is just included by key here, in an attempt to make it more readable.

Snippet 4.6: Assoc for running example Tool-instance

```
assoc(electronicconnection,assoc(ec2,assoc(ebindex,EbindexID,assoc
      ,assoc(pinindex,4,assoc,assoc)),ec1,ec3),assoc,
      assoc(sem,assoc(s1,assoc(eboards,[16_PIN,64_PIN],assoc,assoc),assoc,s2),assoc,assoc))
```

Implementation of Select For Select we want to iterate over our assoc, and build a new assoc based on the result of evaluation on each node in the original assoc. The straightforward method to achieve this might have been to build the new assoc *as we go*. However, since we have opted to go for the non-selfbalancing ALs, it is not possible to build new ALs one node at a time. Instead we will build a Key-Value list, which can then be converted back to an assoc by using the predicate list_to_assoc/2 in the assoc library. The implementation can be found in Snippet 4.7. Line numbers are used to walk through the details.

Snippet 4.7: Implementation of Select for assoc representation

```
1  select_assoc(NavLeft,Rel,NavRight,Assoc,Tool_Instance,NewAssoc):-
2          select_assoc(NavLeft,Rel,NavRight,Tool_Instance,Assoc,List,[]),
3          list_to_assoc(List,NewAssoc).
4
5  select_assoc(Key-Value) --> [Key-Value].
6  select_assoc([]) --> [].
7  select_assoc(_,_,_,_,assoc) --> [],{!}.
8
9  select_assoc(NavLeft,Rel,NavRight,Tool_Instance,assoc(Key,Value,L,R)) -->
10         select_assoc(NavLeft,Rel,NavRight,Tool_Instance,L),
11         select_assoc(Out),
```

```
12          select_assoc(NavLeft,Rel,NavRight,Tool_Instance,R),
13          {check_attribute_assoc(NavLeft,Rel,NavRight,Tool_Instance,Value) -> Out = Key-Value ;
                Out = []}.
```

1-3: This is the entry point predicate. Arguments are exactly the same as for select_list, with of course the exception that ALs are used instead of lists. In line 3 we converted the generated Key-Value list back to an assoc. Line 2 is the entry point to a Definite Clause Grammar (DCGs) based parsing of our assoc. DCG rules are on the form *head - -> body*, which can be read as *"a possible form for head is body"* [28]. We have in the rest of our implementation, four such rules. Note that there is a difference between - -> and -> seen on line 13. The first arrow belongs to the DCG syntax, while the second is special Prolog syntax for a if-then-else statement.

5 (Rule 1): A Key-Value pair is an element in our output list.

6 (Rule 2): The empty list is not an element in our output list. Note that either Rule 1 or Rule 2 is matched through the call on line 11. The argument 'Out' here, is the result of the relation check on line 13. That is, if the check succeeds, the Key-Value pair of the node is passed on, else the empty list is passed on.

7 (Rule 3): The empty node (assoc) is not included in our output list. This rule will match when in line 10 or 11, argument L or R respectively, is the empty node.

9-12 (Rule 4): Our main traversal rule. In 10 we traverse the left subtree, in 12 the right. 10 and 12 are matched by either Rule 3 or Rule 4. In 11 we pass on the result of our 'relation' check in 13, matched by either Rule 1 or Rule 2.

13: Note the use of curly braces here, this allows us to make Prolog calls outside the DCG parsing [28]. We are here taking advantage of this to decide if the relation on the current node holds, in which case we include the node in the output collection. Hence, this is meant to emulate the body of an OCL Select operation.

The Select implementation works as expected, but has the same limitations as for select_list (what kind of Select body we can accept).

Example: Lets say we have the variable EC, instantiated to the full collection of ElectronicConnections in the example Tool-instance. Our selection criteria is the objects with pinIndex = 0. So the OCL would be:

```
ElectronicConnection.allInstances() -> select(e | e.pinIndex = 0) ...
```

Which gives us the following call to select_assoc/6:

```
select_assoc([pinindex],#=,0,EC,Tool_Instance,NewAssoc)
```

We would then get the following answer:

```
PinIndex = 0
NewAssoc = assoc(ec2,assoc(ebindex,2,assoc,assoc(pinindex,0)),assoc,assoc)
```

ec1 and ec3 are not included in $NewAssoc$ since they have pinIndex = 4 and 16 respectively. Hence NewAssoc is a single-node assoc, containing ec2, since its pinIndex could be instantiated to 0.

Implementation of Collect For our implementation of Collect we have a quite similar approach as were done for Select. That is, we will again take advantage of DCGs for traversal, and then convert the resulting list back to an assoc. The implementation can be found in Snippet 4.8.

Snippet 4.8: Implementation of Collect for assoc representation

```
collect_assoc(Nav,Assoc,Tool_Instance,NewAssoc):-
        map_assoc_to_list(navigate_assoc0(Nav,Tool_Instance),Assoc,Result),
        list_to_assoc(Result,NewAssoc).
```

The arguments here are directly comparable to the arguments of collect_list/4, with the exception that ALs are used here. On the second line we are using our version of map_list/3 (in 'lists' library) for ALs, which we have called map_assoc_to_list/3. It has exactly the same functionality as map_list/3, namely having its first argument as a predicate, and applying that predicate to each element/node of its second argument, giving the collection of results in its third argument. We will not give it's full implementation here, for details, see Appendix B. As the input predicate to map_assoc_to_list/3, navigate_assoc0/4 is used, which is the navigation predicate for ALs, returning the full Key-Value pair list. Navigate_assoc/4 on the other hand, only returns the Value list. In Collect we need the pair, since the list afterwards is converted back to an assoc. These navigation predicates will be further explained in subsection 4.2.2.

Example: Given that we have the variable EC instantiated to the ElectronicConnection object of the running example Tool-instance. We now want a collection of all the associated SEM objects. The OCL would be:

```
ElectronicConnections.allInstances() -> collect(e | e.sem) ...
```

Giving is the following call to collect_assoc/4 (EC as second argument):

```
collect_assoc([refs,sem],EC,Tool_Instance,NewAssoc).
```

And Prolog would answer:

```
NewAssoc = assoc(s1,s1_attrs,assoc(s1,s1_attrs,assoc,assoc),assoc(s2,s2_attrs,assoc,assoc))
```

To get the output more readable we have not included the full attribute assoc of our objects. Since both ec1 and ec2 had an association to s1, we have two copies of s1 in NewAssoc. This is however the correct behavior, as the resulting collection of OCL Collect is a bag and not a set (see section 2.2.2.2). This can however cause the new assoc to break the Adelson-Velskii-Landis balance criterion, making subsequent operations on the assoc slower. We have avoided this issue in our experiment, but it might be deserving of attention if one wanted to do a full blown assoc implementation of the OCL operations.

Implementation of IncludesAll Our implementation of IncludesAll can be found in Snippet 4.9.

Snippet 4.9: Implementation of IncludesAll for assoc representation

```
includes_all_assoc(_,assoc):-!.

includes_all_assoc(WholeAssoc,assoc(Key,Value,L,R)):-
        (get_assoc(Key,WholeAssoc,Value2),Value #= Value2 ->
                includes_all_assoc(WholeAssoc,L),
                includes_all_assoc(WholeAssoc,R)).
```

We first see if the second argument can be unified with the empty assoc, in which case the predicate can succeed at once. In the main traversal predicate, the key of the current node is used to check if that key exist in WholeAssoc, with the same value. If this check fails, the predicate can terminate the search at once, since there then exists an example of a node in PartAssoc that does not exist in WholeAssoc. If get_assoc succeeds, the traversal continues in pre-order.

4.2.2 Resolving navigations

As we saw in subsubsection 2.2.2.1, we can navigate associations to refer to other objects and their attributes [17]. We have found two different methods to achieve this in our Prolog implementation. How these two methods are implemented, are again dependent on how the Tool-instance is represented (as discussed in subsection 4.2.1).

We have chosen to call our first method *resolving by containment*, and the second *resolving by id*. These will now be discussed in turn. Our Prolog implementation of the two methods can be found in Appendix B.

4.2.2.1 Resolving by containment

By containment we mean that objects on association ends are copied and stored locally with the association origin. The concept is best explained with an example.

Example: In Figure 4.1, ec1 has an association to s1, lets for now ignore the rest of the elements. If we were to use the list representation of the Tool-instance, with containment, we would get:

```
Tool_Instance = [[[id-1-1,ebindex-EbindexID,pinindex-4,[id-2-1,eboards-[16_PIN,64_PIN]]]], [[
    id-2-1,eboards-[16_PIN,64_PIN]]]]
```

Note here, that s2, with ID = id-2-1, is stored both in the list of SEMs, and together with ec1 (ID = id-1-1). This form of resolving associations has both advantages and disadvantages. On one hand, it should be very fast, regardless of the number of objects in the Tool-instance, since there is a direct access to the association end. On the other hand, increased memory consumption might be an issue, since there can be multiple copies of the same object. But the most severe drawback might be that contained associations puts constraints on what kind of class-models that can be used. Consider for example what would happen if s1 also had an association to ec1, how should this be handled? Without proper handling this would result in infinitely nested objects contained in objects contained in objects etc. For our experiment, the class-model used avoids this issue. But one should consider serious thought before using this sort of association resolving in production code.

4.2.2.2 Resolving by id

With containment we stored an extra copy of the object, when resolving by id, we will in place of this copy instead store an unique identifier. This identifier will then be used to search through the Tool-instance, and retrieve the associated object. Lets reuse the example we had for containment, using id instead:

Example:

```
Tool_Instance = [[[id-1-1,ebindex-EbindexID,pinindex-4,id-2-1]], [[id-2-1,eboards-[16_PIN,64
    _PIN]]]]
```

Since we have the identifier id-2-1, we know that our association end exists in the second object list, as the first object. Note that we do this slightly differently for ALs, here we cannot assume that we know the ordering. Instead we simply use the assigned keys to do our navigation. So instead of for example id-2-1 for lists, you will find something like sem-s1 for ALs. This corresponds to ClassKEY-ObjectKEY. To resolve the association sem-s1, the class key sem are used to find the assoc of sem-objects, and then the object key $s1$ is used to find the correct object within that assoc.

This way of resolving associations avoids the problems that containment had, but it *will* get slower as the number of objects in the Tool-instance grows. However, this directly depends on how the Prolog query is structured. This is thoroughly discussed in section 6.4.

4.2.3 Predicate Organization

Our discussion until now has mainly concerned aspects of the Tool-instance representation in Prolog. This section will talk about something entirely different. Namely, how should we *organize* our Prolog predicates implementing our OCL constraints? The number of possible solutions here are quite possibly unbounded. But we will constrain ourself to two different solutions. These two solutions are identical down to a certain level, so we walk through the organization starting from our entry point predicate; *check_system/1*.

Say that we have the following four OCL constraints (body not shown):

```
Context ElectronicConnection inv ec_1
Context ElectronicConnection inv ec_2

Context SEM inv sem_1
Context SEM inv sem_2
```

The entry point to Prolog is *check_system(Tool_Instance)*. The Prolog representation of the Tool-instance is first structured by class. We will take advantage of this and first extract the respective object collections, as can be seen in the following pseudo-code:

```
check_system(Tool_Instance):-
  get(electronicd_connections, Tool_Instance, ElectronicConnections),
  get(sems, Tool_Instance, Sems),
  check_electronic_connections(ElectronicConnections),
  check_sems(Sems).
```

At this point we can split the four constraints into two sets, each only concerning the objects of its context. Next we can move inside check_electronic_connections.

```
check_electronic_connections(ElectronicConnections):-
  forAll(check_one_electronic_connection,ElectronicConnections).
```

That is, apply the predicate check_one_electronic_connection on each element of ElectronicConnections. This is a pattern we can use when we have OCL constraints of the following type:

```
Type.allInstances() -> forAll(body)
```

That is apply *body* to each element of the collection Type.allInstances(). We have also identified another pattern, where OCL collection operations are used directly on the full collection of a type.

Example:

```
Type.allInstances() -> select(body) ...
```

Here the constraint is based on the full collection of Type.allInstances(), not isolated to one element at a time. With this in place we can now redefine check_electronic_connections to the following:

```
check_electronic_connections(ElectronicConnections):-
  forAll(instance_checks_electronic_connections,ElectronicConnections),
  class_wide_checks_electronic_connections(ElectronicConnections).
```

We have chosen to categorize constraints concerning one object at a time as *instance checks*, while constraints concerning the full object collection as *class wide checks*. These two types of constraints will be further explained when we introduce the constraints we have chosen for our experiment (see section 4.3).

We have now finally arrived at the point where we see two different ways of moving forward. Lets expand *instance_checks_electronic_connections*. Firstly we will see an organization method me have named "normal", while we have named our alternative "condensed". These can be seen in Snippet 4.10 and Snippet 4.11 respectively.

Snippet 4.10: Normal organization

```
instance_checks_electronic_connections(ElectronicConnection):-
  inv_ec_1(ElectronicConnections),
  inv_ec_2(ElectronicConnections).
```

Snippet 4.11: Condensed organization

```
instance_checks_electronic_connections(ElectronicConnection):-
  *body of inv_ec_1*
  *body of inv_ec_2*
```

In Snippet 4.11 we have simply extracted the contents of inv_ec_1 and inv_ec_2. The same organization can also be done for class_wide_checks_electronic_connections. The effects of this reorganization will be thoroughly investigated when we perform our experiment.

Consider however the following case:

> When several constraints operate within the same context, and have overlapping expressions, we can effectively reduce this set of expressions to only being called once when using our condensed predicate organization.

Example: Lets say that we in both inv_ec_1 and inv_ec_2 perform a navigation to the associated SEM object. When these two constraints are condensed, only one navigation is necessary. We have done this filtering manually, but we do believe it is possible to automate the process. One solution could be to first generate the "normal" organization of predicates, and then apply pattern matching on the goals inside each predicate. Then if a goal already exists in the new condensed organization, it can be ignored.

When implementing the constraints of section 4.3, we found several examples where this condensing was possible without affecting the semantics of the constraint.

4.3 Example transformation of OCL constraints

In this section we will show how we manually transformed two sets of OCL constraints. These constraints are written in the context of our example SimPL-model (see Figure 2.5).

The two sets, each contains two constraints. The main difference between the two sets are based on what we found in subsection 4.2.3. That is, we differentiate between constraints applying to only one object at a time, and constraints applying on the full collection of objects of a certain type. These we chose to call *instance checks*, or *class wide checks* respectively.

We will now turn to discuss these two sets.

4.3.1 Set 1: Instance checks

The two constraints can be found in Snippet 4.12 and Snippet 4.13.

Snippet 4.12: Instance 1

```
ElectronicConnection.allInstances()->forAll(ec | ec.pinIndex >= 0 and ec.sem.eBoards->
    asSequence()->at(ec.ebIndex+1).numOfPins > pinIndex)
```

That is, in each ElectronicConnection instance, its pinIndex must be equal to or greater than 0. Moreover, the eBoard at the index equal to ebIndex, must have a value greater than pinIndex. The collection of eBoards is found through navigation on the associated SEM instance.

Snippet 4.13: Instance 2

```
ElectronicConnection.allInstances()->forAll(ec | ec.ebIndex > 0 and ec.ebIndex < ec.sem.
    eBoards->size())
```

In the second constraint we require ebIndex to be greater than 0, and less than the size of the collection of eBoards. We here also include the graphical tree representation, see Figure 4.3.

For our transformation, this tree representations can be seen as the source, while a prolog predicate is the target. We will *not* provide mapping rules for a transformation between the two. We will simply provide what we believe to be a good transformation. This can then be generalized in later work.

4.3.1.1 Transformation

In this section we will show how we transformed the constraint "Instance 2" to a Prolog predicate. We will refer back to the AST in Figure 4.3 to identify which part of the constraint we currently are discussing. The first transformation will be based around the list representation of the Tool-instance. Later, the transformation of "Class wide 2" (see subsubsection 4.3.2.1) will be based on the assoc representation of the Tool-instance.

Snippet 4.14 shows the finished implementation, the line numbering is used to walk through the details. Note that this is slightly simplified compared to the actual implementation.

Snippet 4.14: Instance 2: Prolog implementation

```
1 check_system(Tool_Instance):-
2   nth1(1,Tool_Instance,ElectronicConnections),
3   maplist(instance_2(Tool_Instance),ElectronicConnections).
4
5 instance_2(ElectronicConnection,Tool_Instance):-
```

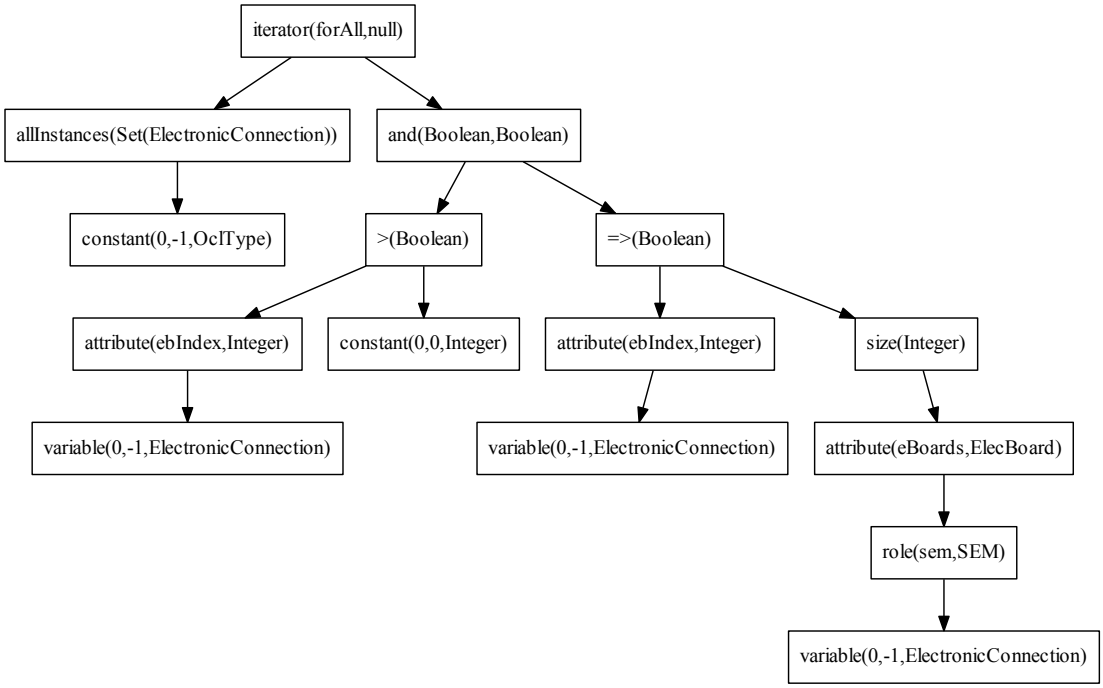

Figure 4.3: Tree representation of Instance 2

```
6         nth1(2,ElectronicConnection,ebindex-EbIndex),
7         navigate_list([refs-4,sem-1,eboards-2],Tool_Instance,ElectronicConnection,Eboards),
8         EbIndex #> 0,
9         length0(Eboards,EboardsSize),
10        EbIndex #=< EboardsSize.
```

1: Entry predicate, containing the full Prolog representation of the Tool-instance

2: We get the collection/list of ElectronicConnections. This corresponds to the following node in Figure 4.3: [allInstances(Set(ElectronicConnection))].

3: We then use maplist/2 to constrain each object in $ElectronicConnections$ using instance_2. This corresponds to the root node: [iterator(forAll,null)]

5: instance_2, with one ElectronicConnection and the Tool-instance as input.

6-7: In our first two goals we get the attribute and association we later need. Note that this does not correspond to the ordering in Figure 4.3. We found it logical to do it this way, but we are unsure if it is viable in an automatic transformation. Some adjustments are sure to be needed.

In line 6 we get the ebIndex attribute using nth1/2. Here we are simply calling nth1, with its first argument equal 2. That is, give me the second element in the list. We can do this since one of our assumptions is that we know the ordering of attributes. This corresponds to the node: [attribute(ebindex,Integer)]. Note that we have two such nodes, but we only need to get the attribute once.

In line 7 we get the eBoards of the associated SEM object. To get to this attribute we must use our specialized predicate for navigating lists, namely navigate_list/4. This corresponds to the nodes: [role(sem,SEM)] and [attribute(eBoards,ElecBoard)].

8: Constrain ebIndex to be larger then 0. This corresponds to the nodes: [>(Boolean)] and [constant(0,0,Integer)], along with what we did in line 6.

9: Get the size/length of the eBoards list we found in line 7. This corresponds to the node: [size(Integer)].

10: Finally, constrain eBindex to be equal to or less than the size of our eBoards list. Corresponding to the node: [=<(Boolean)].

4.3.2 Set 2: Class wide checks

Our two constraints can be found in Snippet 4.15 and Snippet 4.16.

Snippet 4.15: Class wide 1

```
ElectronicConnection.allInstances()->select(c|c.pinIndex = 0)->collect(c|c.sem)->includesAll(
    SEM.allInstances())
```

That is, find me the ElectronicConnection objects that have an pinIndex equal to 0. From this collection, collect all associated SEM objects into a new collection, and make sure that all the SEM objects in the Tool-instance are contained in this collection.

Snippet 4.16: Class wide 2

```
ElectronicConnection.allInstances()->collect(c|c.sem.eBoards)->size() >= SEM.allInstances()->
    collect(c|c.eBoards)->size()
```

That is, collect all owned eBoards, starting navigation from our ElectronicConnection instances. And make sure that the size of this collection is greater than or equal to the collection generated from collecting the eBoards of all SEM instances in the Tool-instance. The graphical tree of this last constraint can be found in Figure 4.4

4.3.2.1 Transformation

In this section we will show how we transformed the constraint "Class wide 2" to a Prolog predicate. We will use Figure 4.4 as reference. In Snippet 4.17 you will find our finished implementation, we will use line numbering to walk through the details. Note that we have again simplified the code.

Snippet 4.17: Class wide 2 Prolog implementation

```
1 check_system(Tool_Instance):-
2   get_assoc(electronicconnections,Tool_Instance,ElectronicConnections),
3   class_wide_2(ElectronicConnections,Tool_Instance).
4
5 class_wide_2(ElectronicConnections,Tool_Instance):-
6         collect_assoc([refs,sem,eboards],ElectronicConnections,Tool_Instance,Eboards),
7         size_assoc(Eboards,Eboards_size),
8         get_assoc(sem,Tool_Instance,SEMs),
9         collect_assoc([eboards],SEMs,Tool_Instance,SEMs_eboards),
```

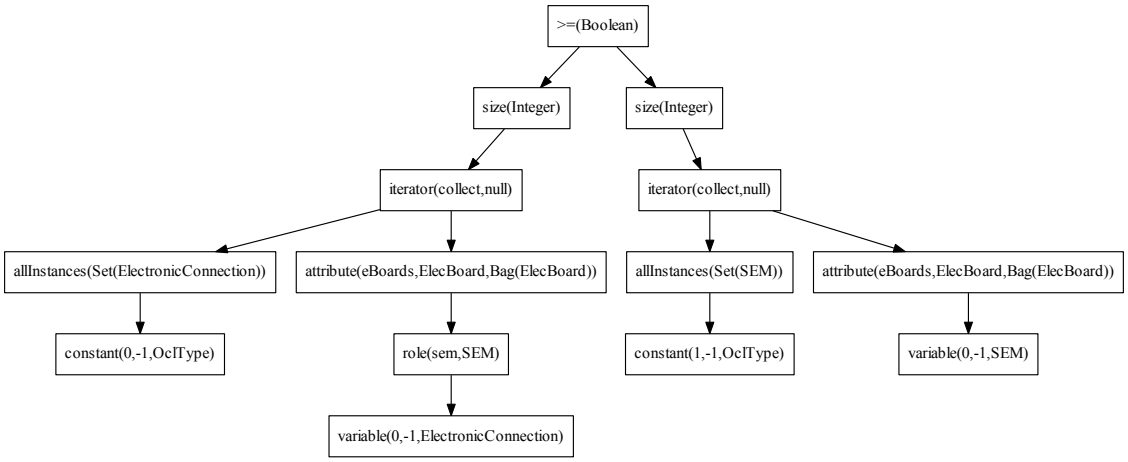

Figure 4.4: Tree representation of class wide 2

```
10          size_assoc(SEMs_eboards,SEMs_eboards_size),
11          Eboards_size #>= SEMs_eboards_size.
```

1: Entry predicate containing the full Prolog representation of the Tool-instance.

2: We first get the collection/assoc of ElectronicConnections using get_assoc/3 ('assoc' library). Note that we can here use the key "electronicconnections" to find the right collection in the Tool-instance. This corresponds to the node: [allInstances(Set(ElectronicConnection))].

3: We pass our collection of ElectronicConnections to class_wide_2.

6: Here we use our special collect predicate for assocs. Collecting the eBoards of the associated SEM objects of all our ElectronicConnections. Corresponding to the nodes: [iterator(collect,null)], [role(sem,SEM)] and [attribute(eBoards,ElecBoard,Bag(ElecBoard))].

7: We then get the size of the collection we found in line 6. Corresponding to the (leftmost) node: [size(Integer)].

8: Here we fetch the collection/assoc of all all SEM objects. Corresponding to the node: [allInstances(Set(SEM))].

9: Collect all eBoards belonging to our collection of SEM objects. Corresponding to the node: [iterator(collect,null)] and [attribute(eBoards,ElecBoard,Bag(ElecBoard))].

10: Get the size of the collection we found in line 9. Corresponding to the (rightmost) node: [size(Integer)].

11: Finally, constrain that the first eBoard collection we found, is greater than or equal to the second collection. Corresponding to the node: [>=(Boolean)].

4.3.3 Complexity

As seen in subsection 2.2.4, constraint complexity is defined as the number of objects that must be accessed to evaluate the constraint. This is represented as a complexity function. We also found that this functions was dependent on the various underlying Prolog implementations. Specifically, which Tool-instance representation, and which navigation method that were used. When we use the 'contained' way of resolving associations, the complexity does not change however (direct access with both Tool-instance representations). So to get the complexity functions of our sets of constraints, there are three different cases to consider:

Contained: The associated objects are contained within the source object. Providing direct access.

ListID: Associations are resolved through IDs, and the Tool-instance is represented as a lists.

AssocID: Associations are resolved through IDs, and the Tool-instance is represented as an assoc.

In Table 4.1 we have summarized the different complexity functions. Note that we have used *ec* as shorthand for electronicConnection. To get these functions we used our implementation of the algorithm presented in [9]. Our implementation can be found in Appendix A.

	Instance Set
List and Assoc - Contained	$4 * P_{ec}$
List and ID	$2 * P_{ec} + 2 * P_{ec} * P_{sem}$
Assoc and ID	$2 * P_{ec} + 2 * P_{ec} * log(P_{sem})$
	Class wide Set
List and Assoc - Contained	$3 * P_{ec} + P_{sem} + P_{ec} * log(P_{sem})$
List and ID	$2 * P_{ec} + P_{sem} + P_{ec} * P_{sem} + P_{ec} * P_{sem} * log(P_{sem})$
Assoc and ID	$2 * P_{ec} + P_{sem} + P_{ec} * log(P_{sem}) + P_{ec} * log(P_{sem}) * log(P_{sem})$

Table 4.1: Complexity functions for our constraints, on the basis of underlying Prolog implementation.

Chapter 5

Evaluation

In the last chapters we have identified several factors that can affect PET. We can put these into the following three categories:

1. Factors coming from transformation choice-points/implementation.
2. Factors coming from SimPL-model/PL-model.
3. Factors coming from Tool-instance/product under configuration.

In category 1, we have complete control of our factors. That is, it is up to us to choose how we want to implement our transformation. And as a result of this, we will consider these our *main factors*, as they are what we are primarily are interested in. Our goal is to find the *levels* of our main factors that provides the lowest PET. By levels of a factor we mean the different variations we have for that factor. For example, one of our main factors is "Predicate organization", where we have two levels; normal, and condensed (see subsection 4.2.3.

While we have full control in category 1, the last two categories we have no way of controlling or constraining. We do of course have control for the purposes of our experiment, but we are here rather referring to the "real life" context. For example, in category 3 we have the factor "Tool-instance size". In our experiment, we have full control over this size (which are the levels of this factor). But conversely, we can not control the size of the Tool-instance outside of the experiment. What we can do however, is to provide estimates on how our code performs on a subset of the available factor levels from these categories. One can consider the combinations of these factor levels as creating different contexts/backdrops for our transformation choices. By performing our experiment we should then get a good understanding of how our different transformations perform in different contexts. Can we find a set of transformations that overall are better than all other sets, or does the context heavily affect PET. We will label the factors in the two last categories as our *extraneous factors* [12]. Our main focus will be on factors coming from transformation choice, but it is also important to uncover how these extraneous factors affect our results.

In the next section, we will summarize the factors of the categories above. We will then, in section 5.2 see how we setup these factors for our experiment. Next, we will in section 5.3 briefly describe how we used an in-house developed tool to carry out our experiment. Finally we will in section 5.4 look at the results of our experiment. These will be further analyzed in chapter 6.

5.1 Factors

In this section we will summarize the main and extraneous factors of our experiment, and give the null-hypothesis for each factor. In general, the null-hypothesis is that there are no relationship between some phenomena [12]. For our context, this means that our base null-hypothesis is that there are no difference between the levels within each factor. That is, varying the levels of a factor, does *not* affect PET.

In addition, we will in subsection 5.1.4 present OCL constraint complexity as a candidate for encompassing several of our factors into *one* factor. That is, the potential for OCL constraint complexity alone, being able to predict much of the variation in PET. Finally, we will in subsection 5.1.5 explain interaction effects, which is the case when the effect of the levels of one factor, is dependent on the levels of another factor [12].

5.1.1 Main factors - from transformation choices

For each factor we will use the following template:

Factor name

H_0 : The null-hypothesis for this factor.

Levels: A description of the different levels our factor will take in our experiment.

Keys: In graphs and data-tables we will use these names to refer to the levels of this factor. One key per level of the factor.

The following factors are related to how the Tool-instance can be represented as a Prolog query, how we choose to represent associations, and finally how we choose to organize our predicates (see the respective subsections in section 4.2. Note that we for the second factor, are using the name "Reference Type", instead of using "association" to coin the term. This is simply to avoid confusion with *association lists*, which is a level of our first factor, "Model Representation"[1].

Factor: Model representation See subsection 4.2.1

H_0 : Whether our Model representation uses lists or association lists, does not affect execution time.

Levels We have implementations for a pure list representation, and an association list representation.

Key list | assoc

Factor: Reference Type See subsection 4.2.2

H_0 : Whether our association representation is done by containment or id, does not affect execution time.

[1] In afterthought, this factor would have been better named "Instance Representation". This was unfortunately not realized before long after all findings-diagrams had been generated using the original name. In the text, we hence will refer to this factor as the Tool-instance representation and Model representation interchangeably.

Levels Associations are either resolved by containment or id.

Key contained | id

Factor: Predicate organization See subsection 4.2.3.

H_0 : Whether our predicate organization is normal or condensed, does not affect execution time.

Levels Normal or condensed

Key normal | condensed

5.1.2 Extraneous factors - from SimPL-model/PL-model

From the SimPL-model we only have one factor, namely the attached constraints. This is primarily a result of the simplification we are operating with, namely constraining any multiplicity to only be 1.

Now, it is important to note that we by treating the set of OCL constraints as a factor, we are not talking about their complexity. OCL constraint complexity as a factor will be discussed in subsection 5.1.4. Here we are merely considering our constraints as different factor levels, and want to see if the different levels affect PET.

In our experiment, we use the OCL constraints presented in section 4.3. Although it is preferable to do the experiments with a wider range of OCL constraints, we deem the OCL constraints presented in section 4.3 sufficient for the purpose of our evaluation, as they provide a representative set of constraints in the domain of ICS configuration. Note that our constraint are written in the context of Figure 2.5.

Factor: Constraint set See section 4.3.

H_0 : Which constraint set we are using does not affect execution time.

Levels Either our set of instance specific constraints, our set of class wide constraints, or both combined.

Key instance | class | combined

5.1.3 Extraneous factors - from Tool-instance/product under configuration

From the Tool-instance there are two extraneous factors; the size of the instance, and the degree of which attributes of objects are configured or not. We define the Tool-instance size as the number of objects in the Tool-instance, hence the numbers of attributes are not taken into account. The Tool-instance used is an instantiation of the Tool-model, transformed from the SimPL-model in Figure 2.5.

To get a fairly realistic picture of how the implementations perform with a large Tool-instance, we will use a size of 40000. That is 20000 ElectronicConnection objects and 20000 SEM objects. In addition we would like to explore how the different factors react to an increase in Tool-instance size, hence we will also perform the experiment with a size of 2000. Note that

we also carried out an experiment with more Tool-instance sizes, to investigate OCL constraint complexity as a factor. The results can be found in subsection 5.4.4.

The other factor is included to investigate the following: A Tool-instance where all attributes are unconfigured marks the start of the configuration process, and when all are configured marks the end. By running the experiment with both of these extreme cases, we hope to uncover if the number of unconfigured attributes affect PET.

Factor: Tool-instance size :

H_0 : Difference in Tool-instance size does not affect execution time.

Levels The Tool-instance will either contain 2000 or 40000 objects.

Keys: 2000 | 40000

Factor: Configured/Unconfigured attributes :

H_0 : Whether our attributes are configured or unconfigured, does not affect execution time.

Levels All attributes are either configured or unconfigured.

Key configured | unconfigured

5.1.4 OCL constraints complexity: One factor to rule them all?

We have dedicated quite some space to discuss OCL constraint complexity (see subsection 2.2.4 and subsection 4.3.3. One of our objectives is to see if we can predict the impact on PET, in large part, just from the complexity of our constraints. If we find support for this, OCL constraints can be abstracted to complexity functions. We could then give general estimates on PET based on the growth rate of these functions; linear growth, quadratic growth etc.

We have reason to believe that OCL constraint complexity can be such an encompassing factor, as it is a direct function of four of our factors. If we have managed to calculate complexity accurately, we could replace these four factors with just one *complexity factor*. The four factors are the following:

I Constraint Set

II Model representation

III Reference Type

IV Tool-instance Size

From the first three factors, we get the complexity function, while the Tool-instance provides the numbers to plug into this function. We have created a separate experiment to investigate our hypothesis about complexity and PET. Here we will use a much larger range of Tool-instance sizes to test the correlation between complexity and PET. A broad overview of our results can be found in subsection 5.4.4, and will be further discussed in chapter 6.

5.1.5 Interaction effects

Until now we have not specifically mentioned interaction effects. But it is a particularly important concept, and one we will dedicate much time to in the section 5.4 and chapter 6. When we use the word *effect*, we use it in relation to our dependent variable, which for us is PET. We have an interaction effect when the effect of the levels of one factor, varies across the levels of another factor [12]. In all, the study of interaction effects can uncover cases such as when a factor only (or to a larger degree) have an effect on PET, when combined with other factors.

Example: Say we wanted to investigate the effects of coffee and sleep on productivity. We could have the following setup:

Dependent variable - Productivity: written words per hour

Factor 1 - Coffee: 0 - 5 cups today

Factor 2 - Sleep: 0 - 8 hours of sleep last night

Is a person drinking coffee more productive than one that does not? Is a sleepy person less productive then one that is fully awake? If we only looked at the effect of these factors in isolation, we might find that both drinking coffee and having enough sleep correlates positively with productivity. However, if we included an analysis of interaction effects, we might find that drinking coffee only effected productivity when paired with a sleepy person.

Like done for the individual factors, the null-hypothesis will also be defined for the interaction effects. First we have the interaction between main factors:

I There is no interaction effect between Model Representation and Predicate Organization.

II There is no interaction effect between Reference Type and Model Representation.

III There is no interaction effect between Reference Type and Predicate Organization.

Interaction between extraneous factors:

I There is no interaction effect between Number of Objects and and Configuration.

II There is no interaction effect between Number of Objects and Constraint Type.

III There is no interaction effect between Configuration and Constraint Type.

There can of course also exist interactions between main and extraneous factors as well. In fact, there exist a potential of 9 additional interactions between the two groups of factors. These will be discussed in subsection 5.4.3.

5.2 Full factorial design

For our experiment we will utilize a so called full factorial design. This means that all combinations, of all the levels of all our factors, are present in our experiment [31]. This is important as we then have a design where all the *main factors* are grouped with all levels of the *extraneous*

factors. Hence we should not have any confounding[2] variables affecting the findings. With this setup we can later use ANOVA (Analysis of Variance) to detect which of our factors, and which interactions, are significant. That is, whether we can accept or reject the null-hypothesis for each of our factors. ANOVA will be discussed more in section 6.1.

As seen in section 5.1, we have a total of 6 factors. All factors have two levels, with the exception of "Constraint Set", which has three. We will call one combination of our factors a *setup*. The number of possible setups then becomes $2 * 2 * 2 * 2 * 2 * 3 = 96$. The result of running one setup we call an *observation*. For each setup, we will generate 10 observations. This gives us a total of 960 experiment runs. In Table 5.1 an excerpt of our setups can be found. Here we have not included the factor "Number of objects[3]", to minimize the size of our table.

#	Cons.	Ref. Type.	Model Rep.	Conf.	Pred. Org.
1	instance	containment	list	configured	normal
2	instance	containment	list	configured	condensed
3	instance	containment	list	unconfigured	normal
4	instance	containment	list	unconfigured	condensed
5	instance	containment	assoc	configured	normal
6	instance	containment	assoc	configured	condensed
7	instance	containment	assoc	unconfigured	normal
8	instance	containment	assoc	unconfigured	condensed
9	instance	id	list	configured	normal
10	instance	id	list	configured	condensed
11	instance	id	list	unconfigured	normal
12	instance	id	list	unconfigured	condensed
13	instance	id	assoc	configured	normal
14	instance	id	assoc	configured	condensed
15	instance	id	assoc	unconfigured	normal
16	instance	id	assoc	unconfigured	condensed
17	class	containment	list	configured	normal
18	class	containment	list	configured	condensed
19	class	containment	list	unconfigured	normal
20	class	containment	list	unconfigured	condensed
21	class	containment	assoc	configured	normal
22	class	containment	assoc	configured	condensed
23	class	containment	assoc	unconfigured	normal

Continued on next page

[2]Two variables/factors are confounded when we have combined them in such a way that their effects can't be separated [12].

[3]i.e., the Tool-instance size

Table 5.1 – continued from previous page

#	Cons.	Assoc.	Model Rep.	Conf.	Pred. Org.
24	class	containment	assoc	unconfigured	condensed
25	class	id	list	configured	normal
26	class	id	list	configured	condensed
27	class	id	list	unconfigured	normal
28	class	id	list	unconfigured	condensed
29	class	id	assoc	configured	normal
30	class	id	assoc	configured	condensed
31	class	id	assoc	unconfigured	normal
32	class	id	assoc	unconfigured	condensed
33	combined	containment	list	configured	normal
34	combined	containment	list	configured	condensed
35	combined	containment	list	unconfigured	normal
36	combined	containment	list	unconfigured	normal
37	combined	containment	assoc	configured	condensed
38	combined	containment	assoc	configured	normal
39	combined	containment	assoc	unconfigured	condensed
40	combined	containment	assoc	unconfigured	normal
41	combined	id	list	configured	normal
42	combined	id	list	configured	condensed
43	combined	id	list	unconfigured	normal
44	combined	id	list	unconfigured	condensed
45	combined	id	assoc	configured	normal
46	combined	id	assoc	configured	normal
47	combined	id	assoc	unconfigured	condensed
48	combined	id	assoc	unconfigured	normal

Table 5.1: The different setups used in our experiment (Tool-instance size not included)

In the next section we will briefly describe how we setup our experiment with the setups from Table 5.1.

5.3 Experiment setup with PrologQueryGenerator

PrologQueryGenerator (PQG) was written in an effort to automate the process of running all our different setups, and capturing the results of those setups. We will not go into much im-

plementation details concerning PQG, but source code is available at `https://bitbucket.org/ThomasRolfsnes/rolfsnes_thesis`. We will now in broad strokes explain the process we went through to setup our experiment. In section 4.3 we showed how we created a set of different Prolog implementations, depending on our different transformation choices. Our next step was then to generate queries that could be evaluated using our different implementations. The structure of these queries depended on the following factors:

- How we represent the Tool-instance (2 levels).
- And how associations should be represented (2 levels).

This gave us a total of four different transformations that were implemented with PQG. In addition, the configuration of attributes, and the size of the Tool-instance, needed to be controlled. This were achieved in PQG by creating "ExperimentSetup" objects. With this in place it was just a matter of creating the 48 setups found in Table 5.1, and evaluating them with both Tool-instance sizes. Evaluation was done using Jasper [8], which provides us with a bidirectional interface between SICStus Prolog and Java. To get the Prolog execution time, the built in statistics/2 predicate were used.

```
statistics(runtime, [T0|_]),
*query-goes-here*,
statistics(runtime, [T1|_]),
ExecutionTime is T1 - T0.
```

The value of 'ExecutionTime' were then retrieved. This value along with all other factor levels for each setup were then written to a CSV file. The full data-set can be found at `https://bitbucket.org/ThomasRolfsnes/rolfsnes_thesis`.

Details about the hardware/software that were used to run the experiment can be found in Table 5.2

Parameter	Value
CPU	i5-3570K CPU @ 3.40GHz (4 CPUs)
RAM	8192MB RAM
Operating System	Windows 8 Pro 64-bit
Prolog Interpreter	SICStus Prolog 4.2.3

Table 5.2: Hardware/Software of computer running the experiment.

5.4 Findings

In this section we will give a broad overview of our findings, before we in chapter 6 will perform a deeper analysis of the data using ANOVA. We will first look at the *main effects* of our main factors. That is, what was the mean PET[4] for any given levels of our factors. For example, to calculate the main effect of having Reference Type equal 'containment', take the PET for

[4]A reminder: This is our abbreviation for Prolog Execution Time.

all setups where we used containment, and calculate the mean [12]. In addition we will also look at the two-way interaction effects between our main factors. After looking at the results concerning our main factors, we will do a similar summary for our extraneous factors. Finally, we will in subsection 5.4.4 look at the results of our separate experiment, where we investigate the possibility of having OCL constraint complexity as a main factor.

5.4.1 Main factors

We will start the summarization of our data by looking at the *main effects* of our main factors. The results can be found in Figure 5.1. The horizontal line in the middle is the overall mean for all setups.

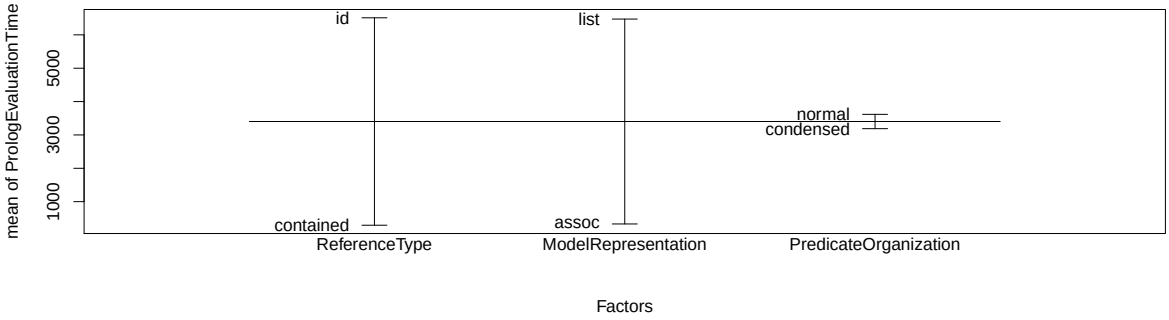

Figure 5.1: The main effects of our main factors

Figure 5.1 will give us an idea of which of the factors that played the biggest part in affecting PET. We see that changing the levels of 'ReferenceType' and 'ModelRepresentation' had a large impact on the results, while how predicates were organized seemingly played a lesser role. The actual PET means for each factor level can be found in Table 5.3.

Factor	Levels	Mean evaluation time ($\approx$ in ms)
Reference Type	id	6511 ms
	contained	293 ms
Model Representation	list	6473 ms
	assoc	331 ms
Predicate Organization	normal	3616 ms
	condensed	3187 ms

Table 5.3: Mean evaluation times for our factors

Only measuring main effects might not give the full picture though (see subsection 5.1.5). Potential interaction effects between the factors must also be measured. To investigate this, interaction plots can be used. An interaction plot shows how different combination of levels of two factors interact [32]. If the lines in an interaction plot cross (are not parallel), we have an

indication of an interaction effect [12]. But to be certain, we have to test the significance of the interaction, which we will do in chapter 6.

For our three main factors we will need a total of 3 plots to get all the combinations. In Figure 5.2 we have the interaction plot for the factors: 'ModelRepresentation' and 'PredicateOrganization'. We see that there is a slight angle between the lines here, but there probably is no interaction effect. Then in Figure 5.3 we have the interaction plot for 'ReferenceType' and 'ModelRepresentation'. Here it is quite clear that we probably have an interaction effect. While both Tool-instance representations gives similar PETs when using contained associations, the list representation perform substantially worse when using id associations. Finally in Figure 5.4, we have the interaction plot for 'ReferenceType' and 'PredicateOrganization'. Which we see in similar to the plot in Figure 5.2. And again, probably no interaction effect.

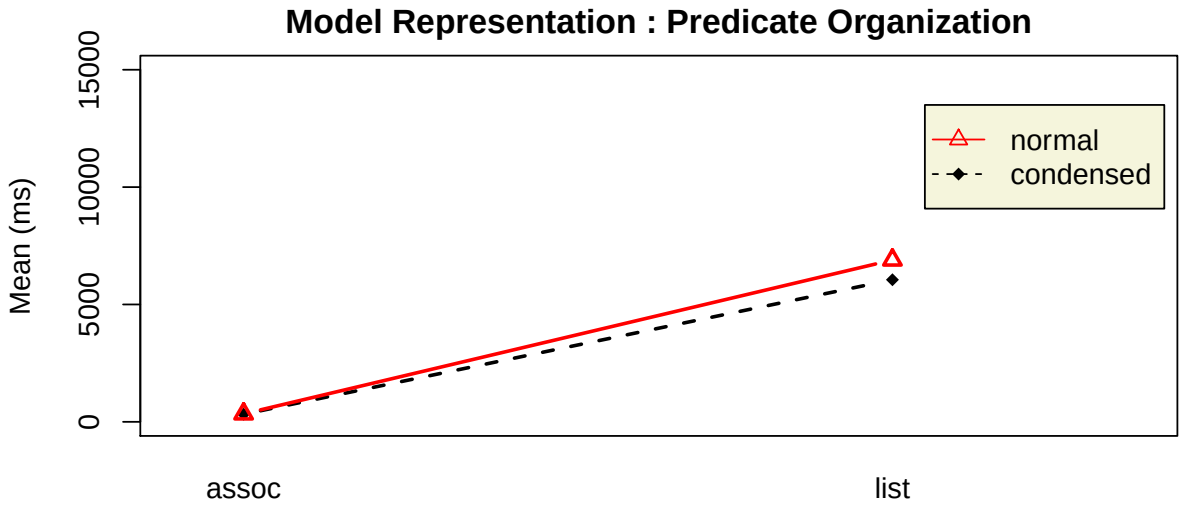

Figure 5.2: Interaction plot for 'ModelRepresentation' and 'PredicateOrganization'.

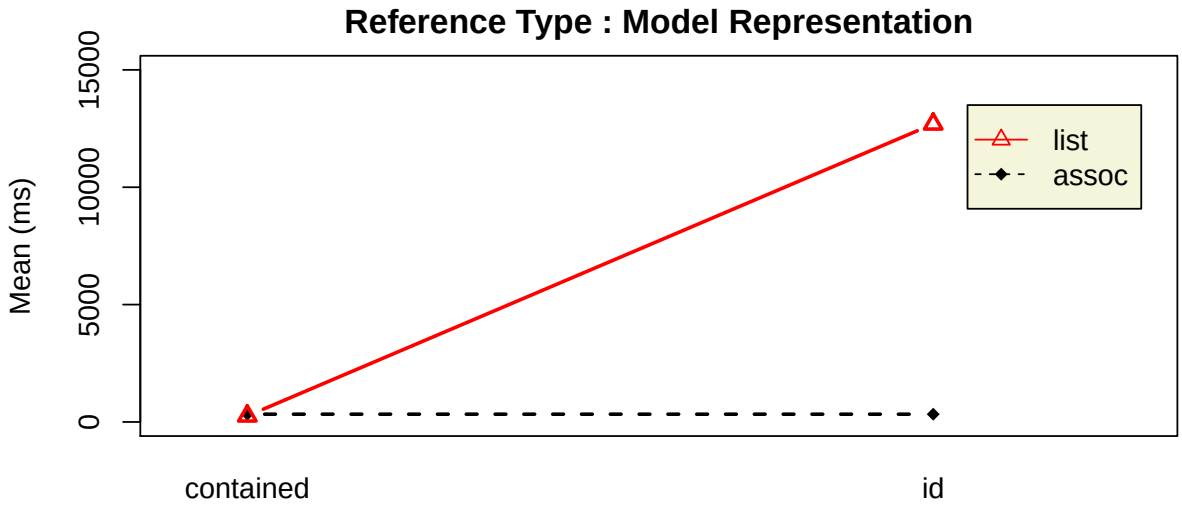

Figure 5.3: Interaction plot for 'ReferenceType' and 'ModelRepresentation'.

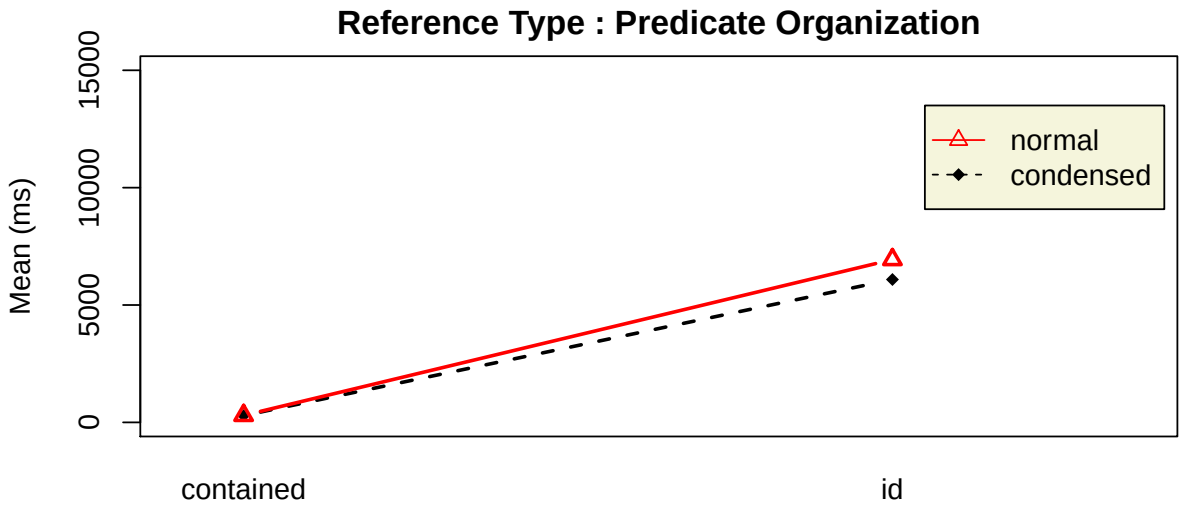

Figure 5.4: Interaction plot for 'ReferenceType' and 'PredicateOrganization'.

Looking at our interaction plots, we probably only have a significant interaction effect between which Tool-instance representation we are using, and how we represent associations. This is a great example of why it is important to also consider interaction effects. If we were to only look at main effects, we might write of id-associations as leading to high PET. What we find here however, is that this only happens when paired with the list representation. We will look more into this relationship in chapter 6.

5.4.2 Extraneous factors

We will now take a brief similar look at our extraneous factors. In Figure 5.5 you will find the main effects of these factors.

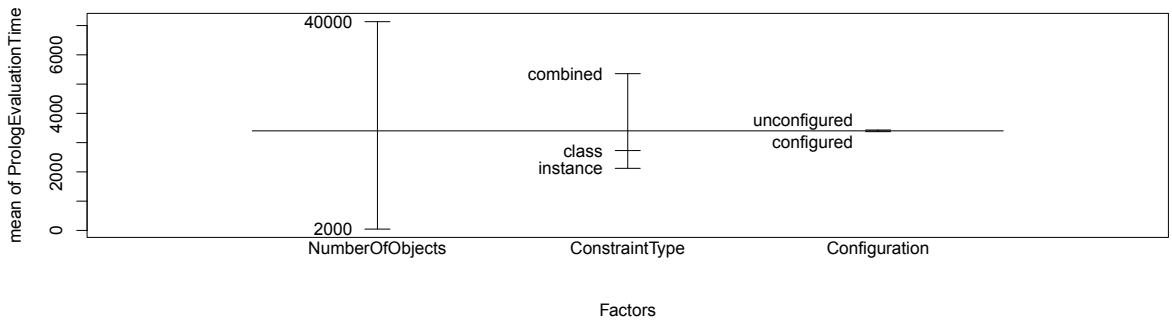

Figure 5.5: Main effects of our extraneous factors.

In Figure 5.5 we see that Tool-instance size apparently plays a large role in determining PET. We also see that our different constraint sets performed differently. Finally we see, somewhat

surprisingly, that the configuration of attributes seemingly does *not* impact PET. The actual PET means can be found in Table 5.4.

Factor	Levels	Mean evaluation time (≈ in ms)
Number of objects	2000	44 ms
	40000	7132 ms
Constraint type	class	2729 ms
	instance	2121 ms
	combined	5356 ms
Configuration	configured	3374 ms
	unconfigured	3430 ms

Table 5.4: Mean evaluation times for our extraneous factors

Lets now look at the interaction plots for our extraneous factors. Again we will need three plots. In Figure 5.6 we have the interaction plot for 'NumberOfObjects' and 'Configuration'. Here we see two almost perfectly parallel lines, giving us a strong indication that there are no interaction effects here. Then in Figure 5.7, we have the interaction plot for 'NumberOfObjects' and 'ConstraintType'. We see that the size of the Tool-instance clearly interacts with which constraint set we were using. Finally, in Figure 5.8, we have the interaction plot for 'Configuration' and 'ConstraintType'. Here we see that changing attribute configuration does not interact with which constraint set we are using.

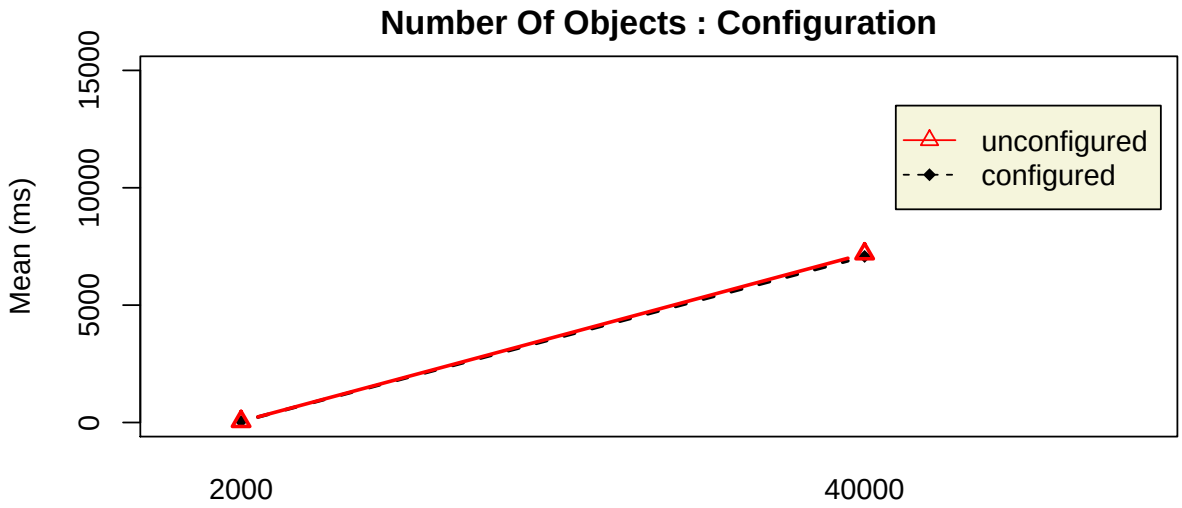

Figure 5.6: Interaction plot for 'NumberOfObjects' and 'Configuration'.

We have now measured main effects and interaction effects separately for our main and extraneous factors. In the next section we see if there are any interesting interaction between our two categories of factors.

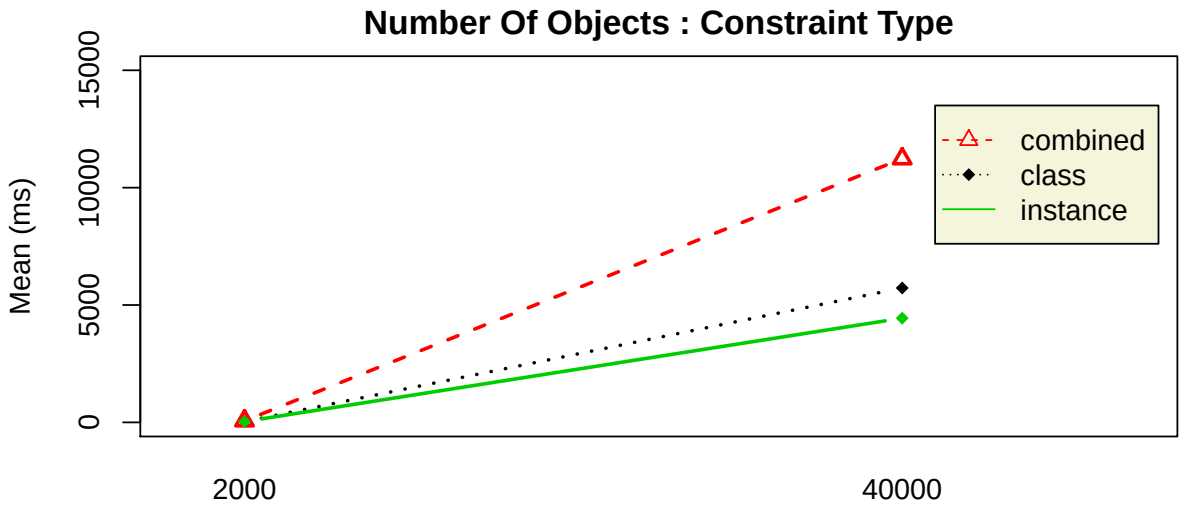

Figure 5.7: Interaction plot for 'NumberOfObjects' and 'ConstraintType'.

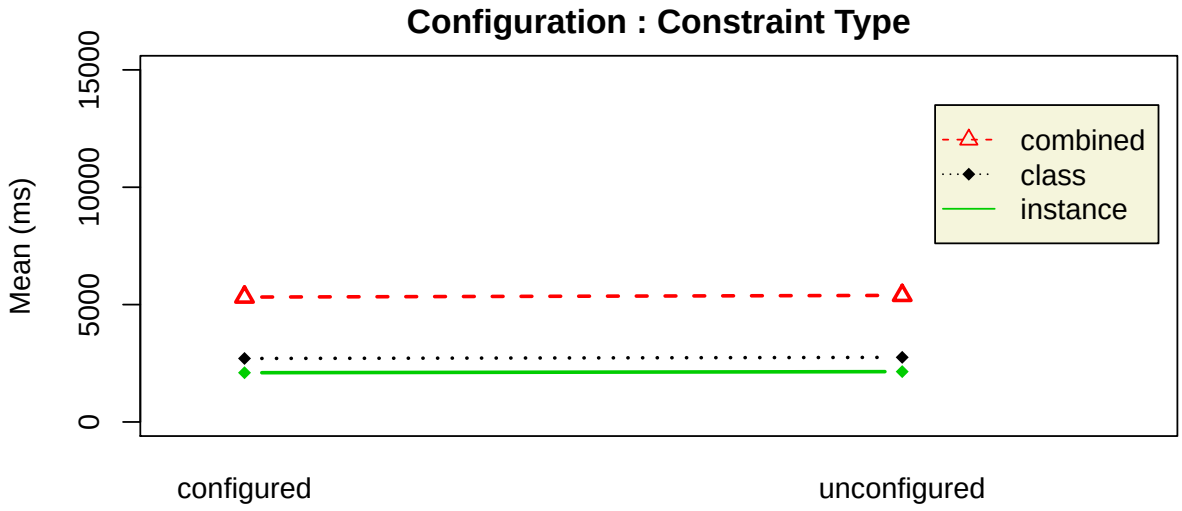

Figure 5.8: Interaction plot for 'Configuration' and 'ConstraintType'.

5.4.3 Other interactions

In addition to the interaction plots found in subsection 5.4.1 and subsection 5.4.2, we did also generate plots to investigate interactions between our two categories of factors. From these plots we only found one to be interesting, namely the interaction between constraint sets and predicate organization. The plot can be found in Figure 5.9. Here we see that while our 'combined' and 'class' sets of constraints leads to similar PET when changing attribute configuration, the 'instance' constraint set achieves slightly lower PET when using condensed organization of predicates. We will look more into this in section 6.2.

5.4.4 OCL constraint complexity as a main factor

This section will investigate the potential for treating OCL constraint complexity as a fully encompassing factor. That is, can complexity be used to be the sole predictor of PET? This

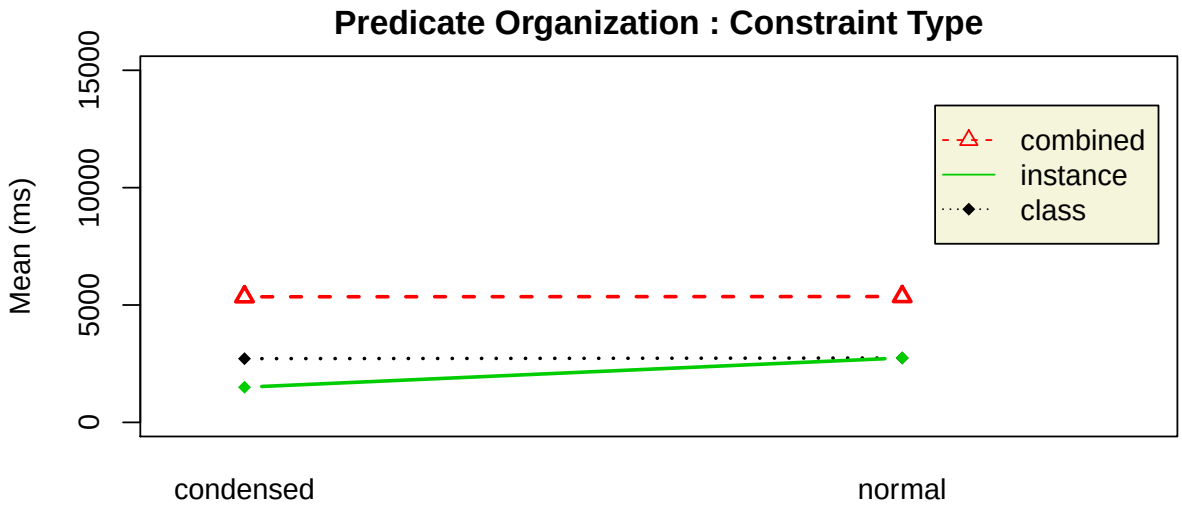

Figure 5.9: Interaction plot for 'PredicateOrganization' and 'ConstraintType'.

translates to the following null-hypothesis: H_0 : *There is a 1 to 1 correlation between OCL constraint complexity and PET.* To test this hypothesis, we have to measure if an increase in complexity leads to a correlated increase PET. To get a proper data-set, we have augmented the experiment setup explained in section 5.2. The factor 'NumberOfObjects' will now take on the following values: 1000 | 3000 | 5000 | 7000 | 9000 | 11000. Remember that Tool-instance size instantiates the parameters of the complexity functions. Varying Tool-instance size should hence provide a good spread in complexity from low to high.

An important note is that the 'condensed' predicate organization has been filtered out of the resulting data-set. As explained in subsection 4.2.3, some navigation calls could be removed when condensing the predicates. Doing this invalidates the relationship between the complexity function of a set of OCL constraints, and their implementation as Prolog predicates. That is, the condensed version of a set of predicates no longer contains all the constructs of the original OCL constraints, hence rendering the complexity function invalid.

Figure 5.10 plots OCL constraint complexity against PET, using the data-set described above. It is evident that there is *not* a 1 to 1 relationship here, but we clearly see that there are three trends. We will further analyze this plot in section 6.3.

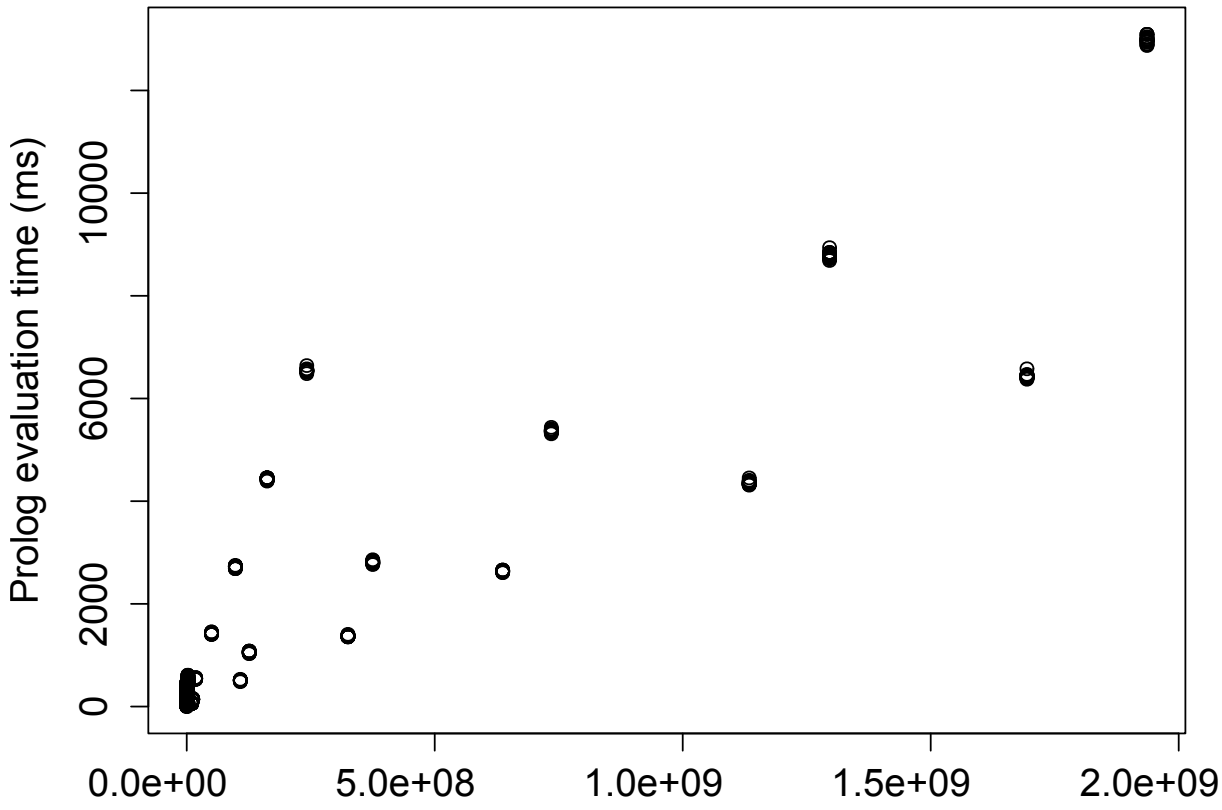

Figure 5.10: Plot of OCL constraint complexity and PET.

Chapter 6

Analysis and Discussion

In this chapter we will start by giving a brief explanation of ANOVA, which is the main statistical test our analysis is based on. We will then apply ANOVA to the results found in section 5.4. Here we found indications of the factors 'ReferenceType', 'ModelRepresentation', 'NumberOfObjects' and 'ConstraintType' providing main effects. 'PredicateOrganization' and 'Configuration' apparently did not. In addition we found indications of the following interaction effects: 'ReferenceType:ModelRepresentation', 'NumberOfObjects':'ConstraintType' and 'PredicateOrganization:ConstraintType'. By applying ANOVA we can test the significance of these results. That is, are the effects found due to random chance, or can they be explained by the factors? Based on what is found here, we will then find the combinations of factor levels that should provide the lowest PET. In section 6.3 we will look at the results of our separate OCL complexity experiment, before we in section 6.4 discuss our method and potential threats to validity.

6.1 ANOVA

ANOVA, or "Analysis Of Variance" is a statistical test, that unsurprisingly, is for *"analyzing the variance that appear in data"*[12]. That is, it can be used to test if a factor is a significant provider to the observed variance. Conversely, a factor can also be found to *not* provide a significant effect on the observed variance. Technically, it divides the variation in the effect/dependent-variable into groups. The groups are based on *"which factors that are assumed to be responsible for that variation"* [12]. These groups are then used to calculate a so called F-ratio. The F-ratio are used to determine if the variation produced by the different levels of a factor, is significant to a given significance-level. In the analysis presented in section 6.2, a significance level of 0.01 will be used. That is, if a factor is found to be significant, there is less than a 1% chance that the result is due to chance. In this case, we can reject our null-hypothesis. That is, that the average mean of each factor level is the same. On the contrary, we can then say that the different levels of a factor provide different PET means.

However, in [12], it is suggested to take extra care if an interaction effect is present. If a factor is found to have a significant main effect, we are saying that regardless of other factors, this factor will have an effect on the result. When a significant interaction effect is found however, this might not be true anymore. In [12], it is recommended to avoid interpreting main

effects when an interaction effect is present. We will follow this guideline in the next section.

6.2 ANOVA based analysis

We will start by looking at the ANOVA table for the main factors subsection 5.1.1, this can be found in Table 6.1. The syntax factorA:factorB refers to the interaction effect between factorA and factorB. Note that the three way interaction between the main factors have also been included.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	Sig.
ModelRepresentation	1	8601613043.37	8601613043.37	142.13	0.0000	***
ReferenceType	1	8815432704.63	8815432704.63	145.66	0.0000	***
PredicateOrganization	1	41983658.96	41983658.96	0.69	0.4051	
ModelRep.:Ref.Type	1	8818940035.11	8818940035.11	145.72	0.0000	***
ModelRep.:Pred.Org.	1	38848442.96	38848442.96	0.64	0.4232	
Ref.Type:Pred.Org.	1	40354553.69	40354553.69	0.67	0.4144	
ModelRep.:Ref.Type:Pred.Org.	1	39624177.20	39624177.20	0.65	0.4186	
Residuals	904	54711297301.75	60521346.57			

Significance codes: 0: '***', 0.001: '**', 0.01: '*', 0.05: '.'

Table 6.1: ANOVA table for main factors

Table 6.1 confirms our findings from subsection 5.4.1. By reading the right most column (Sig.) we find both 'ModelRepresentation' and 'ReferenceType' to be highly significant, which means that the different levels of these factors provide main effects on PET. But further, the interaction effect between these two factors were also found to be highly significant. As discussed in section 6.1, we will therefore focus on this interaction effect. Neither 'PredicateOrganization', nor the other interactions, were found to be significant.

If the case had been that there were *no* interaction effects, a combination of the factor levels with lowest PET, would have produced the overall lowest PET. However, since we have an interaction effect, all possible combinations of factor levels, of the participating factors, must be assessed. In Figure 6.1 this situation has been plotted as a tree. We first branch on the two levels of 'ReferenceType', and then on each node we again branch on the two levels of 'ModelRepresentation'. This gives us four possible combinations. On each leaf node there is a boxplot showing the full range of PET that were measured under that combination of factor levels.

When using 'contained' references, 'assoc' and 'list' yields similar PET. When using 'id', 'assoc' still provides low PET, while combining with 'list' gives a large spread in PET. By examining the data-set, the id-list interaction was found to be highly sensitive to Tool-instance size and which constraint set that had been used. This probably explains the large spread. With the three other combinations performing well, we conclude that any of these can be chosen

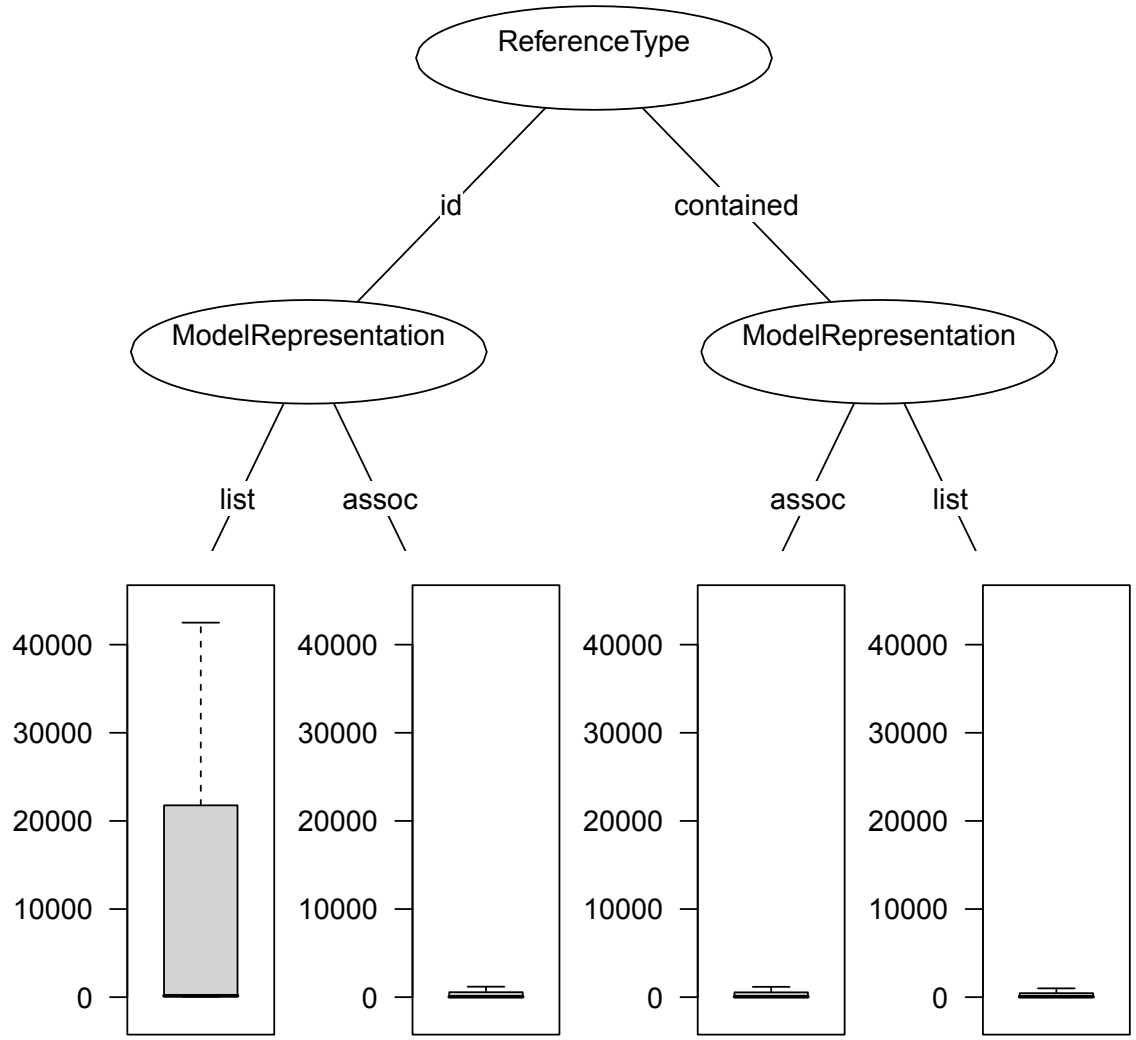

Figure 6.1: PET spread for the interaction between ReferenceType and ModelRepresentation. Numbers are PET in milliseconds.

to achieve low PET. There is however some restrictions that apply to using 'contained' references, as earlier discussed in subsubsection 4.2.2.1. Since it was found that 'id' references also can achieve low PET, when paired with the right Tool-instance representation, 'contained' references should not be preferred.

We will now move on to look at the extraneous factors:'NumberOfObjects','Configuration' and 'ConstraintType' The ANOVA table can be found in Table 6.2.

In Table 6.2 a similar situation as for the main factors are found. Two factors, 'NumberOfObjects' and 'ConstraintType' were found significant, as well as their interaction. Neither 'Configuration', nor the other interactions were found significant. While investigating which factors levels of the interaction that gives the lowest PET would be possible, it's usefulness is limited. What we rather can learn from this is:

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	Sig.
NumberOfObjects	1	11422659686.67	11422659686.67	155.92	0.0000	***
Configuration	1	714000.36	714000.36	0.01	0.9214	
ConstraintType	2	1797172126.16	898586063.08	12.27	0.0000	***
Num.Obj.:Conf.	1	707649.76	707649.76	0.01	0.9217	
Num.Obj.:Cons.Type	2	1953514910.00	976757455.00	13.33	0.0000	***
Conf.:Cons.Type	2	30611.78	15305.89	0.00	0.9998	
Num.Obj.:Conf.:Cons.Type	2	28876.79	14438.39	0.00	0.9998	
Residuals	900	65933266056.17	73259184.51			

Significance codes: 0: '***', 0.001: '**', 0.01: '*', 0.05: '.'

Table 6.2: ANOVA table for extraneous factors.

I It does not matter how many attributes of the object are unconfigured/configured. Hence users of our configuration tool should experience stable performance through the configuration process.

II A Tool-instance with many objects does not necessarily impact PET.

III Neither does the constraints.

IV It is the interaction between the size of the Tool-instance, and which constraints that are used, that is important.

See for example Figure 6.2. Since the 'list-id' factor level combination earlier was found to heavily impact PET negatively, it has here been removed from the data-set to show optimal PET. Here we clearly see that Tool-instance size interacts with the constraint sets differently.

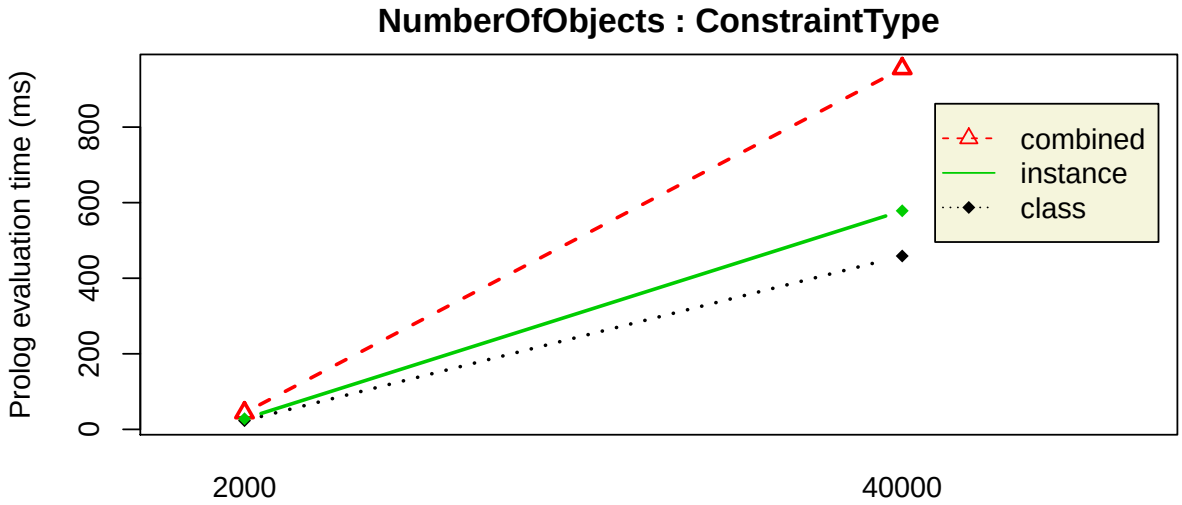

Figure 6.2: Interaction effect between 'NumberOfObjects' and 'ConstraintType', given optimal combination of main factors.

In subsection 5.4.3 a potentially interesting interaction was found between 'PredicateOrganization' and 'ConstraintType', it seemed that condensing the organization of predicates only had effect on one of the constraint sets. By further investigation, the cause proved to be obvious. One of the main advantages of condensing the predicates, was that duplicate navigation calls could be removed. The 'instance' constraint set (Snippet 4.12 and Snippet 4.13), which showed lower PET with condensed predicates, had some navigation calls removed. While the other set (Snippet 4.15 and Snippet 4.16), had not. Further experiments showed that this was indeed the cause for the change in PET. Now, in Table 6.3 you will find the ANOVA table for this interaction.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	Sig.
PredicateOrganization	1	41983658.96	41983658.96	0.48	0.4885	
ConstraintType	2	1797172126.16	898586063.08	10.28	0.0000	***
Pred.Org.:Const.Type	2	76376071.12	38188035.56	0.44	0.6462	
Residuals	906	79192562061.43	87409008.90			

Significance codes: 0: '***', 0.001: '**', 0.01: '*', 0.05: '.'

Table 6.3: Interaction effect between 'PredicateOrganization' and 'ConstraintType' showing no significant interaction as a result of biased data.

Notice that no significant interaction was found between 'PredicateOrganization' and 'ConstraintType', even though we know from the previously described experiment that there is an interaction here. This exemplifies the importance of not trusting statistical tests blindly, and being aware of the details of the data-set you're working with. It is quite possible that the interaction would have been found significant, if even more navigations had been removed from the constraint sets. In that case, 'PredicateOrganization' also probably would had been found to be a main effect.

6.3 Analysis of OCL complexity results

In subsection 5.4.4 it was found that there were no 1 to 1 relationship between OCL constraint complexity and PET. There were however three trends in the plot in Figure 5.10. That is, the data-points seemed to diverge in three different directions. The data-set used has only one factor with three levels, namely 'ConstraintSet'. It was therefore natural to suspect this factor as the cause of the three trends. In Figure 6.3 the different constraint sets has been plotted separately. We see that our assumption that the factor 'ConstraintSet' was responsible for the three trends, appears to be correct. But still there is a concentration of data-points to the far bottom-left of each plot, that appears to not follow the trend of the remaining points. In fact, by further investigation we found that to get a 1 to 1 linear trend, one had to filter out the levels of the factors: 'ConstraintSet', 'ReferenceType and 'ModelRepresentation'. That is, all three factors that are used to generate the complexity function. To exemplify, see Figure 6.4, where the following setup has been used:'instance','id' and 'list'. Similarly high correlation plots for other setups were also evident.

To summarize, the predictive capabilities of OCL constraint complexity, are fairly limited. The desired use-case was that complexity could be generalized such that higher complexity implied higher execution times[1], independent of which complexity function that were used in generating the complexity. To enable use of complexity as a predictor for PET, one has to actually run the finished implementation with differing Tool-instance sizes. The data-points produced could then be used in a linear regression to predict PET with larger Tool-instances. At this point however, Tool-instance size is probably an equally good predictor in it self. Although non-linear regression must then be used in cases where we have non-linear growth in PET. See for example Figure 6.5, showing the quadratic growth resulting from combining 'list' model representation with 'id' reference type.

[1] Assuming that the evaluation was done on the same machine of course.

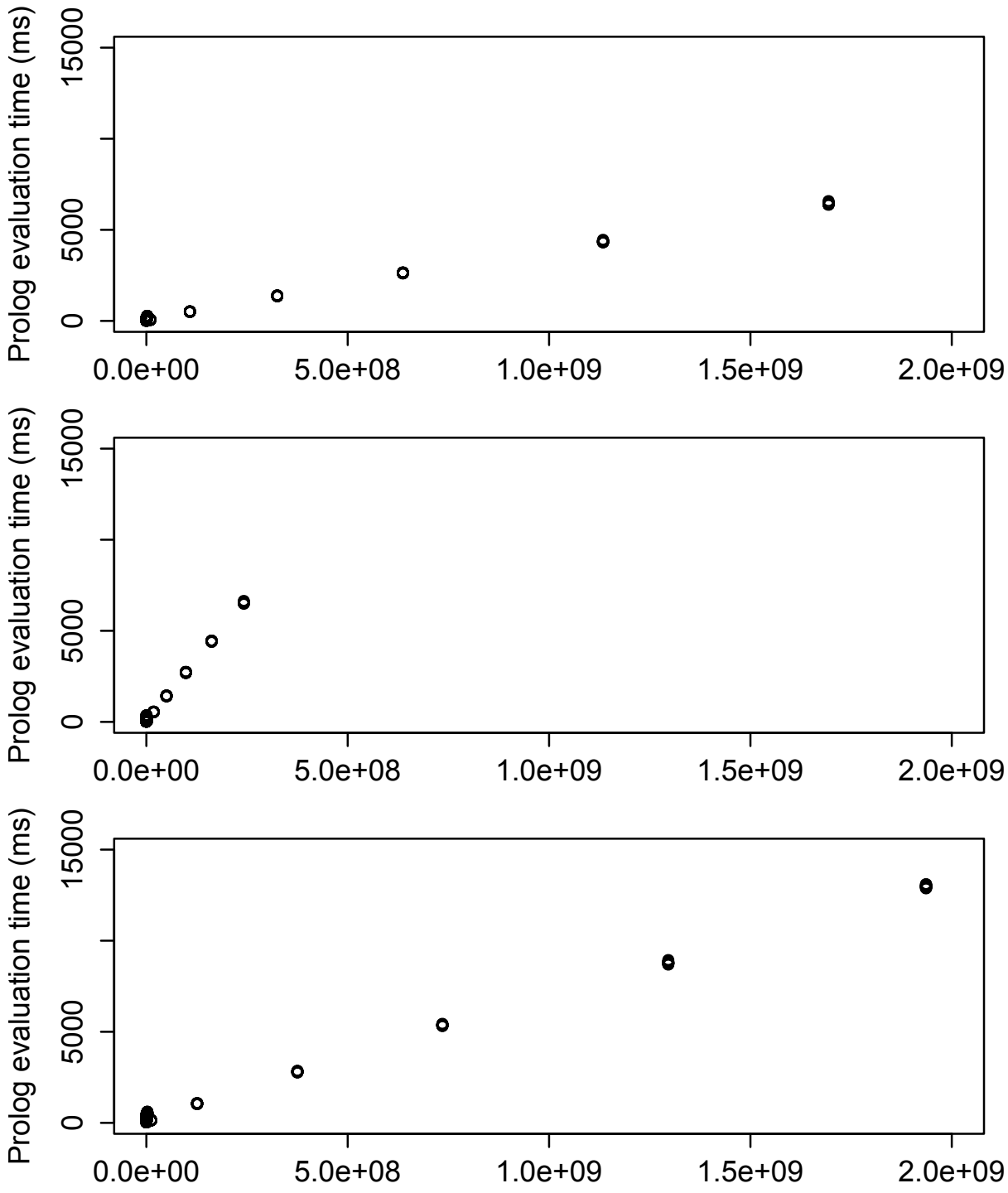

Figure 6.3: Plot of OCL constraint complexity against PET. One plot per constraint set.

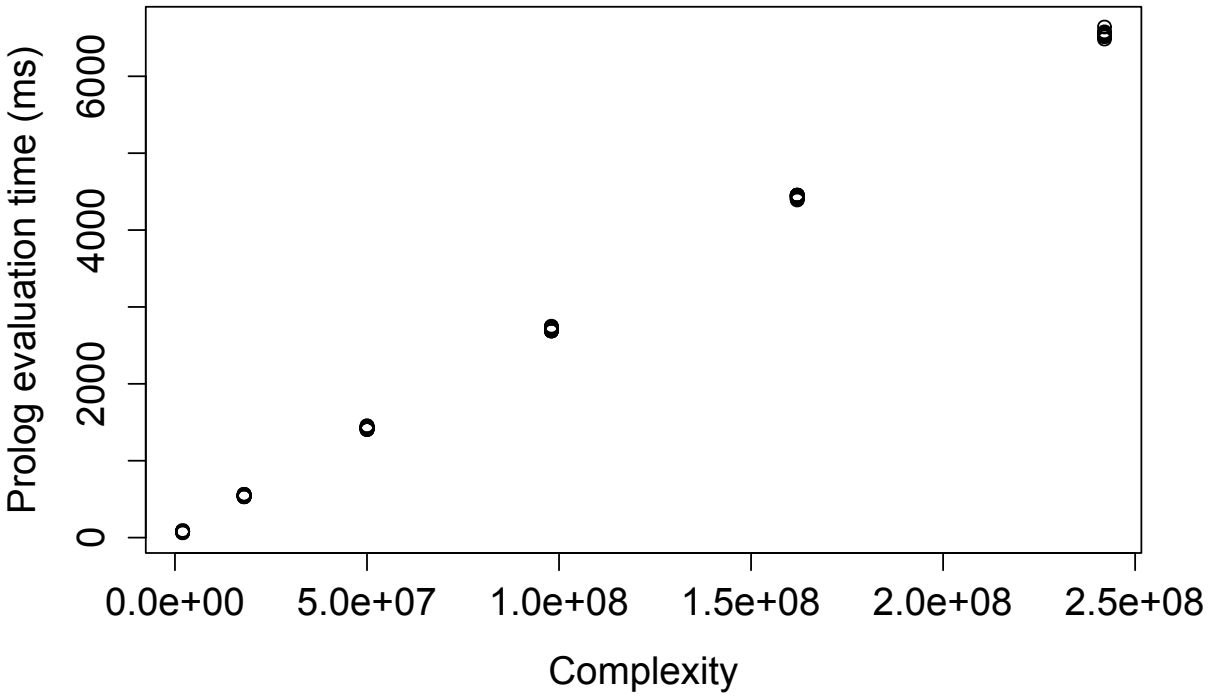

Figure 6.4: Complexity plot for filtered data-set: 'ConstraintSet', 'ReferenceType and 'Model-Representation'.

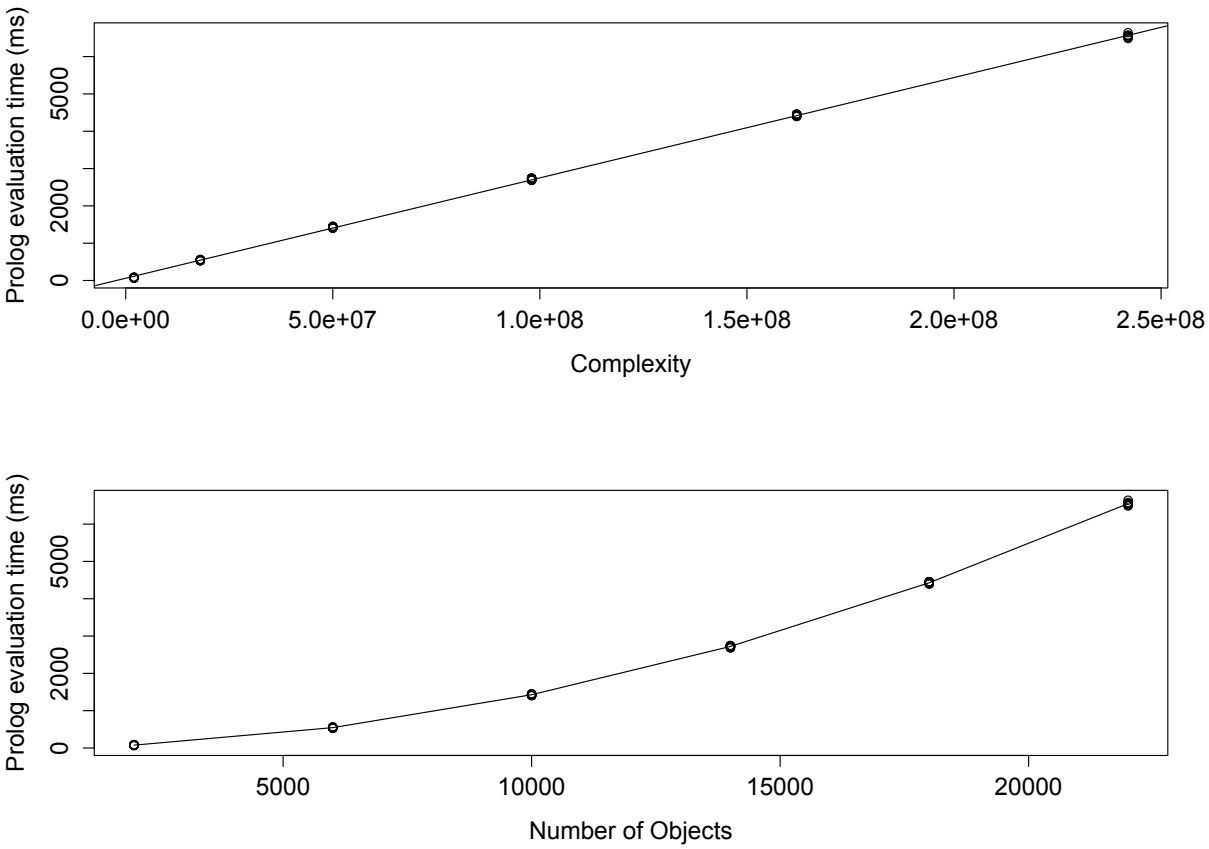

Figure 6.5: Potential for linear regression with complexity, vs non-linear regression with Tool-instance size.

6.4 Discussion

In section 6.2 we presented the following findings:

I It was found that representing the Tool-instance as association lists, paired with 'id' representation of references, gave the lowest PET. Using 'contained' references also gave low PET, but was not chosen as it has problems with generalizing to other class-models (see subsubsection 4.2.2.1).

II It was found that condensing predicates can lead to lower PET in the case that the individual predicates has the same navigations. That is, only one has to be kept when creating the condensed predicate.

III It was found that the interaction between Tool-instance size and the constraints that were used, was more important for PET than the two factors individually.

IV It was found that the amount of configured vs unconfigured attributes in the Tool-instance did not impact PET.

V Finally, in the analyses of OCL constraint complexity as a predictor for PET, it was found that complexity was a non satisfactory predictor.

Our method was based on a small excerpt of a Product Family Model (see Figure 2.5), and two sets of OCL constraints. The constraints were manually implemented in Prolog, with different implementations for the factor levels that were to be investigated. Several different Tool-instances were then generated and evaluated towards the predicates, measuring how long the evaluation took in each case. One possible shortcoming of our method is that our implemented Prolog versions of the included OCL operations might be unequally optimized. For example, the Collect implementation for association lists might be 'better' than the Collect implementations for lists, therefore biasing finding I. As for finding II, even though it was showed that condensing predicates can lead to lower PET, we are uncertain how difficult it is to implement this type of OCL2Prolog transformation in practice. That is, implementing the transformation such that several constraints can share the same navigation calls. Increasing the performance of navigation calls will also decrease the advantage of condensed predicates. Regarding finding III and IV, we see no issues with being generalizable outside our context. In finding V, we concluded that OCL constraint complexity was a non satisfactory predictor for Prolog execution time. However, the validity of the generated complexity functions are based on the assumption that they properly reflect the implementation in Prolog. There is of course the scenario that this assumption is false, in which case complexity still has a potential of becoming a PET predictor.

Finally, we are uncertain to the degree of which our findings are generalizable to other Product Family Models and OCL constraints. Firstly, multiplicity of associations/references were not tested with any other than 1. Secondly, OCL operations did not have generalized implementations in Prolog. That is, operation bodies were constrained to consist of simple navigations and attribute comparisons. Now, we highly doubt that these shortcomings impact the significance of our findings. That is, association lists with id-references will always be better than lists with id-references. The question rather is, if association lists with id-references still perform

satisfactory when paired with Tool-instances containing objects with higher multiplicities. Furthermore, if OCL operations with more complex bodies, and larger sets of OCL constraints in general are included, will this invalidate our findings concerning Tool-instance representation? These question are left for future work.

Chapter 7

Related Work

We will in this chapter look at related work, for this we will only concern ourself with the UML/OCL2Prolog transformation, as it is where our main problem domain resides. Our work on this transformation has had several aspects:

I Creating a transformation from UML/OCL to Prolog

II Specifically, we wanted to find a mapping from a Tool-instance[1] to a Prolog query, and the combined mapping of a Tool[2]-model + OCL constraints to a set of Prolog predicates. Moreover, this should be executable as a CSP.

III Our main concern has been to find the representations that yielded the lowest PET.

IV Finally we wanted to investigate if OCL constraint complexity can be a reliable predictor of PET.

The existing work on UML to Prolog transformations, is mainly focused on enabling querying of models [33] [34] [35]. And specifically, class-models. In [35], the transformation was done by mapping the UML meta-model to an intermediate meta-model (MoMat). All model-elements were then finally mapped to Prolog facts. Queries could then be written to compute different aspects of the class-model. For example counting the number of associations, finding the classes that has exactly 3 attributes, etc. We do believe that representing the Tool-instance as a set of Prolog facts, can be an interesting direction to investigate. MoMat could then probably be adapted to fit our needs. The main advantage we see with using Prolog facts over lists/trees, is in relation to navigation resolution. With lists and trees, this can be done in O(n) and O(log(n)), respectively. This difference in cost of resolving navigations showed an evident impact in chapter 6. Resolving a navigation using Prolog facts on the other hand, can potentially be much faster. In SICstus at least, access to predicates/facts are done using a hash-table, where the facts are indexed on their first argument[3][28]. Adapting this representation such that it is suitable for CSPs, might be a promising line of research. Ideally, one could then lessen the impact of navigations on PET even more.

[1] i.e., real-world-realization of a UML class-model.

[2] i.e., UML class-model

[3] The first argument must be instantiated and unique for this to work. Hence a unique identifier should be used as the first argument.

The work we found that was most aligned with ours was [18]. Here the transformation of both UML and OCL are addressed. Their focus is to test the *satisfiability* of models. That is, if *"it is possible to create a correct and non-empty instantiation of the model"*[18]. In their developed tool[36], a user can specify bounds for the number of objects of a certain class, valid instantiation of attributes of objects, and number of instanced associations. Given a set of transformed OCL constraints, the tool will then determine if these constraints hold within the bounds specified by the user, and return a valid instantiation of the class-model. Labeling[4] is used to find an instantiation of all attributes, that is, their full possible domain is not returned, which is our desired use-case. To our understanding, their instance representation is very much similar to our list representation (see subsubsection 4.2.1.1), using ids to resolve navigations. OCL is transformed using ASTs, into a set of Prolog predicates mimicking the structure of the AST. That is, each node in the AST is transformed to a separate predicate, and evaluation starts from the root node, calling the predicates representing the left/right node in the AST and so on. In our manual transformation of OCL, not much consideration has been put towards how an automatic approach would have been implemented. We saw in section 6.2, that condensing several OCL constraints into one predicate can lead to lower PET. However, if we later realize that our approach is not viable for automatic transformation, [18] should provide good guidance in terms of how it can be done the *normal* way.

We have been unable to find any related work that addresses our main research question, investigating the PET of various representations of UML/OCL in Prolog. The only vaguely relevant study is [37], which evaluates the PET of the model representation found in [35]. This does however provide us with the opportunity to compare our implementation with lowest PET, with the implementation using Prolog facts in [35]. At least we can get an indication of how our implementation compares. Execution time was in [37] measured under several conditions, varying model size, and which queries that were used. To compare with their finding, we will use the same variation in model size, and the one query they used that also are applicable in our context. Namely counting the number of elements in the model. It must be noted that in their model representation, they are counting the number of elements (represented as individual facts), where the elements can either be classes, Associations, Properties and so on. We will count the equivalent amount of *objects*. The prolog code we are running for each model-size can be found in Snippet 7.1. We will for this small experiment use the tree based representation of the Tool-instance (association lists).

Snippet 7.1: Counting the number of objects in the Tool-instance

```
count_objects(Tool_Instance,Size):-
        get_assoc(electronicconnection,Tool_Instance,EC),
        get_assoc(sem,Tool_Instance,SEM),
        size_assoc(EC,ECSize),
        size_assoc(SEM,SEMSize),
        Size is ECSize + SEMSize.
```

Luckily, [37] also includes their implementation, this can be found in Snippet 7.2.

Snippet 7.2: Counting the number of elements. Fact based implementation

```
size(Num) :-
findall(Id,me(_,_),IdList),
```

[4]See section 2.4

```
length(IdList,Num).
```

The resulting plot can be found in Figure 7.1. As it is evident from the plot, we get a significantly faster execution than the fact based model presented in [35]. This should however not be too surprising. In our implementation, we readily have access to the collection of objects we are interested in. And most of the execution time is just spent counting the number of objects. In the fact based model however, the wanted elements must first be collected using the findAll predicate. Our guess is that this is a linear operation. One should not, however, take this as an indication that our representation is better than a fact-based representation. This was a special case where we had a near optimized data-structure for the problem. We believe a more thorough comparison of the two representations can be an interesting line of research. Particularly, if the fact based representation could be adapted to our problem domain of CSPs.

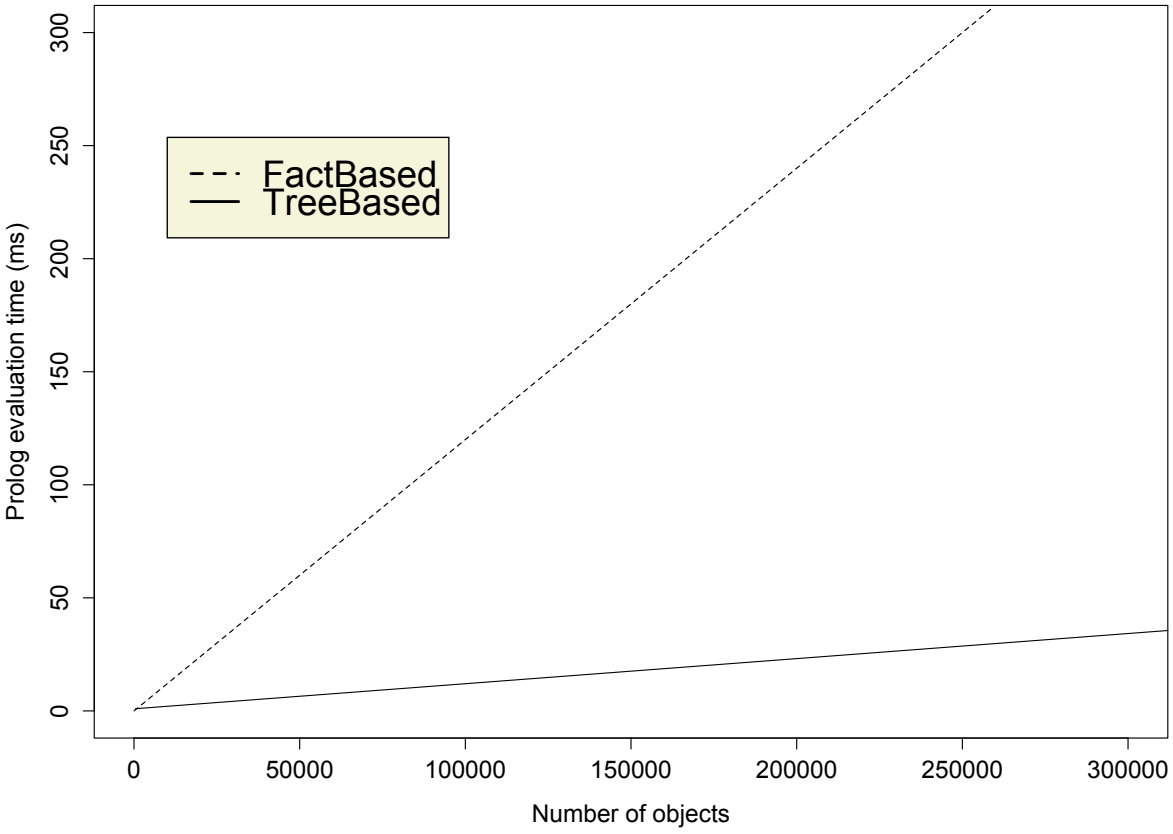

Figure 7.1: Counting the number of elements. Fact-based VS tree-based.

The last research question we proposed in the beginning of this chapter, was if OCL constraint complexity could be a reliable predictor of PET. To the best of our knowledge, this has not earlier been investigated in the literature.

Chapter 8

Conclusion

We have investigated how a product-line, given as a SimPL-model (with OCL), can go through two transformation steps and be represented as a logic program in SICStus Prolog. An instance of the product-line (i.e., product-under configuration/Tool-instance), can then iteratively be evaluated in the logic program, which we have termed the *configuration process*. Our focus has been on the second transformation step (Tool-model+OCL and Tool-instance, to Prolog), where we have investigated several different mappings choices, which we have called *choice points*, or *main factors* in our experiment. In addition we investigated factors that we are unable to control outside the experiment context. These we called *extraneous factors*. We performed an exhaustive investigation of these factors, through experimentation and later data-analyzes using ANOVA. On the basis of this analysis, we presented guidance on which mapping choices that should be performed to achieve the lowest Prolog execution time, based on the factors investigated.

8.1 Main findings

We found that there were significant *interaction effects* between some factors, that is, they could not be considered in isolation. For the main factors, we concluded that representing the Tool-instance as association lists (i.e., a tree structure), combined with resolving associations through reference-ids, yielded the lowest Prolog execution time. We also found that representing associations through *containment* (i.e., the copies of the associated object were stored together with the association source), would give low PET regardless of Tool-instance representation. However, we concluded that the limitations behind doing this outweighed the advantages. Further, we found that condensing predicates (i.e., OCL constraints that had the same context, could be combined and transformed to a single predicate), could lead to lower PET in cases where the individual OCL constraints had the same navigations. That is, we could lower the number of navigations performed. For the extraneous factors, we found a significant interaction effect between Tool-instance size, and which constraints that were used. That is, a big Tool-instance does not necessarily imply high PET, but is largely dependent on which constraints that are used. Further, we found that number of unconfigured attributes (i.e., variables in the Prolog query representation), did not impact PET. That is, there were no significant difference in PET when all attributes were configured, versus when all were unconfigured. Finally, in our inves-

tigation of OCL constraint complexity as a predictor for Prolog execution time, we found no significant correlation between the two.

8.2 Future research

In section 6.4, we acknowledged that there were some questions of validity to our method. Specifically the lack of investigating the effects of associations with higher multiplicities, and the generality of our OCL-operation implementations in Prolog. In addition we acknowledged in chapter 7, that there exists other solutions to both instance[1], and class-model[2]/OCL, representations in Prolog that can be investigated. Future research should address these questions before implementing a full scale version of the second transformation step.

8.3 Concluding Remarks

Product-line engineering is an interesting and blooming field. There are however some caveats. The configuration process can become time consuming and error prone. With the contributions of this thesis, we hope that we have come one step closer in resolving these issues. We firmly believe that with further research, a fully functional configuration tool can be realized.

[1] i.e., real-world-realization of the model
[2] e.g., the Tool-model

Appendix A

Java implementation of OCL constraint complexity algorithms

To be able to measure the complexity of our OCL expressions, we have implemented the algorithm found in [9]. This algorithm does not encompass the whole of OCL, and we also wont expanded the scope in our implementation. However, there is one exception, as we have included support for the OCL operation *includesAll()*. The implementation of some methods and classes used below, will not be repeated here. Full source code can be found at `https://bitbucket.org/ThomasRolfsnes/rolfsnes_thesis`, along with the the implementation of the PrologQueryGenerator.

As reference we first present the original algorithm from [9], in Snippet A.1. Our Java implementation can be found in Snippet A.2. In afterthought, using EOS [16] as our OCL parser probably was not optimal. EOS trimmed away a lot of the information on nodes. We for example had to introduce a small hack to infer if a node was in the left subtree of a loop operation (done with "node.referredVariable.loopExpr" in the original algorithm).

Snippet A.1: Original complexity algorithm

```
Complexity(node: OCLExpression, aux:Value) : Value
  cL:=0; cR:=0;
  if (node.leftChild<>null) cL := complexity(node.leftChild);
  if (node.rightChild<>null) cR := complexity(node.rightChild);
  if (node instanceof LiteralExp) {aux:=0; compl:=0;}
  if (node instanceof VariableExp) { aux:=1;
    if (node.referredVariable.loopExpr<>null) compl:=1
    else compl:=0;}
  if (node instanceof AttributeCallExp) {
     aux:=node.leftChild.aux; compl:=cL;}
  if (node instanceof IteratorExp) {
       aux:= node.leftChild.aux;
       compl:= cL + node.leftChild.aux x cR; }
  if (node instanceof AssociationEndCallExp) {
    if(node.referredAssociationEnd.multiplicity='1') {
       aux:=node.leftChild.aux;
       compl:=cL+ node.leftChild.aux; }
    else{
     aux:= node.leftChild.aux x Nnode.referredAssociationEnd.name;
     compl:= cL + node.leftChild.aux x Nnode.referredAssociationEnd.name ; }
    }
  if (node instanceof OperationCallExp)
    if (node.name='allInstances') {
       aux:= Pnode.type ; compl:=Pnode.type}
```

```
    else  // operations like '+', '-', 'and'
      {aux:=0; compl:=cL + cR; }
return compl;
```

Snippet A.2: Java implementation of OCL complexity algorithm

```
/**
   * @param node The node to calculate the complexity for. Start at root to get complexity for
       full expression.
   * @param gv
   * @param referedVariableLoopExp2
   * @return a simple complexity object
   * @throws Exception
   */
  public Complexity complexity(Node node, REFTYPE refType, boolean referedVariableLoopExp)
      throws Exception {
    // Complexity of this node
    Complexity cThis = new Complexity("", "");
    // Complexity of the left subtree
    Complexity cL = new Complexity("0", "");
    // Complexity of the right subtree
    Complexity cR = new Complexity("0", "");

    String type = "";
    /**
     * Check what the supertype of our node is (the node is a subtype, we
     * only care about the supertype)
     */
    try {
      type = getInstanceExpType(node);
    } catch (Exception e) {
      e.printStackTrace();
    }

    Node leftChild = null;
    Node rightChild = null;

    try {
      leftChild = node.getChildren().get(0);
    } catch (ArrayIndexOutOfBoundsException e) {}

    try {
      rightChild = node.getChildren().get(1);
    } catch (ArrayIndexOutOfBoundsException e) {}

    /**
     * FINISHED SETUP, START COMPLEXITY CALCULATION
     */
    if (leftChild != null && type.equals(IteratorExp)) {
      cL = complexity(leftChild,refType,true);
      } else if(leftChild != null && referedVariableLoopExp){
        cL = complexity(leftChild,refType,true);
      } else if(leftChild != null ){
        cL = complexity(leftChild,refType,false);
      }
    if (rightChild != null) {cR = complexity(rightChild,refType,false);}

    switch (type) {
    case LiteralExp:
      cThis.setAux("0");
      cThis.setCompl("0");
      break;
    case VariableExp:
      cThis.setAux("1");
      if (referedVariableLoopExp) {
        cThis.setCompl("1");
```

```
      } else {
        cThis.setCompl("0");
      }
      break;
    case AttributeCallExp:
      cThis.setAux(cL.getAux());
      cThis.setCompl(cL + "");
      break;
    case IteratorExp:
      cThis.setAux(cL.getAux());
      cThis.setCompl(cL + "+" + cL.getAux() + X + cR);
      break;
    case AssociationEndCallExp:
      switch (refType) {
      case CONTAINED:
        if (isMultiplicityOne(node)) {
          cThis.setAux(cL.getAux());
          cThis.setCompl(cL + "+" + cL.getAux());
        } else {
          cThis.setAux(cL.getAux()+ X + N + getAssosiationEndName(node));
          cThis.setCompl(cL + "+" +  cL.getAux() + X + N
              + getAssosiationEndName(node));
        }
        break;
      case LISTID:
        if (isMultiplicityOne(node)) {
          cThis.setAux(cL.getAux());
          cThis.setCompl(cL + "+" + cL.getAux() + X + P + getType(node));
        } else {
          throw new Exception("List-id: no implementation for higher multiplicities yet");
        }
        break;
      case ASSOCID:
        if (isMultiplicityOne(node)) {
          cThis.setAux(cL.getAux());
          cThis.setCompl(cL + "+" + cL.getAux() + X + "log(" + P + getType(node) + ")");
        } else {
          throw new Exception("Assoc-id: no implementation for higher multiplicities yet");
          }
        break;
      }
      break;
    case OperationCallExp:
      if (node.getNameOperation().equals("allInstances")) {
        cThis.setAux(P + getType(node));
        cThis.setCompl(P + getType(node));
      } else if (node.getNameOperation().equals("includesAll")) {
        cThis.setAux(cL + X +"log(" + cR +")");
        cThis.setCompl(cL + X +"log(" + cR +")" );

      } else { // operations like '+', '-', '*', '<', '>=','and' etc
        cThis.setAux("0");
        cThis.setCompl(cL + "+" + cR);
      }
      break;
    }
    return cThis;
  }
```

Appendix B

OCL in Prolog API

In this appendix you will find the full API we wrote in Prolog to implement the OCL constraints presented in section 4.3.

```
/* -*- Mode:Prolog; coding:iso-8859-1; -*- */


:- module(rolfsnes, [
        navigate_assoc/4,               %Key x ModelAssoc x Assoc -> Assoc
        navigate_assoc0/4,              %Key x ModelAssoc x Assoc -> Key-Assoc
        navigate_list/4,                %Type-Index x Model x List -> Element
        nested_list_to_assoc/2,         %List -> Assoc
        collect_assoc/4,                %NavigationList x Assoc x ModelAssoc -> NewAssoc
        collect_list/4,                 %NavigationList x List x Model -> NewList
        select_assoc/6,                 %NavigationLeft x Rel x NavigationRight x Assoc x
            ModelAssoc -> NewAssoc
        select_list/6,                  %NavigationLeft x Rel x NavigationRight x List x Model
             -> NewList
        includes_all_assoc/2,           %WholeAssoc x PartAssoc -> Boolean
        includes_all/2,                 %WholeList x PartList -> Boolean
        map_assoc_to_list/3,            %Pred x Assoc -> List
        size_assoc/2,                   %Assoc -> Size
        check_attribute_assoc/5,        %NavigationLeft x Rel x NavigationRight x AssocModel x
             Assoc
        check_attribute_list/5,         %NavigationLeft x Rel x NavigationRight x Model x List
        element0/3,                     %Integer x List -> Element
        length0/2,                      %List -> Size
        domain_interval/2,              %List x Interval
        is_pair_list/1                  %List -> Boolean
                    ]).

:- use_module(library(clpfd)).
:- use_module(library(lists)).
:- use_module(library(aggregate)).
:- use_module(library(assoc)).


/*
PROLOG MODEL GENERATION
*/

%@ Adapted from code written by Richard A. O'Keefe, from the SiCStus assoc library
%@  @item list_to_assoc(@var{+List}, @var{-Assoc})
%@  @PLXindex {list_to_assoc/2 (assoc)}
%@  is true when @var{List} is a proper list of @var{Key-Val} pairs (in any order)
%@  and @var{Assoc} is an association tree specifying the same finite function
%@  from @var{Keys} to @var{Values}.  Note that the list should not contain any
%@  duplicate keys.  In this release, @code{list_to_assoc/2} doesn't check for
%@  duplicate keys, but the association tree which gets built won't work.
```

```
%@
%@ This version has been augmented to build association lists from lists of key-value pairs
%@ nested within the value of other key-value pairs. So we will get a tree where the nodes
%@ also can be trees.
nested_list_to_assoc(List, Assoc) :-
        (is_pair_list(List) ->
        keysort(List, Pairs),
        length(Pairs, N),
        nested_list_to_assoc(N, Pairs, [], Assoc)
        ;
        Assoc = List).

nested_list_to_assoc(0, List, List, assoc) :- !.
nested_list_to_assoc(N, List, Rest, assoc(Key,Val2,L,R)) :-
        A is (N-1) >> 1,
        Z is (N-1) - A,
        nested_list_to_assoc(A, List, [Key-Val|More], L),
        (nonvar(Val) -> nested_list_to_assoc(Val,Val2) ; Val2 = Val),
        nested_list_to_assoc(Z, More, Rest, R).
/*
NAVIGATION
*/

%@ @item navigate_assoc(@var{KeyList},@var{Model},@var{Assoc},@var{Result})
%@
%@ Navigates @var{Assoc} according to the list of navigation steps given in @var{KeyList}.
%@ Just returns the value of the found node
navigate_assoc([Key],_,Assoc,Element):-
        get_assoc(Key,Assoc,Element).

navigate_assoc([refs,ClassKey|Rest],Model,Assoc,Result):-
        get_assoc(refs,Assoc,Refs),
        get_assoc(ClassKey,Refs,Object),
        (is_assoc(Object) ->
                (is_multiplicity_one_assoc(Object) -> get_first_key_value_assoc(Object,_,
                    Element))
                ;
                get_assoc(ClassKey,Model,ClassList),
                get_assoc(Object,ClassList,Element)),
        (Rest == [] -> Result = Element ; navigate_assoc(Rest,Model,Element,Result)).

navigate_assoc([Key|Rest],Model,Assoc,Result):-
        get_assoc(Key,Assoc,Element),
        navigate_assoc(Rest,Model,Element,Result).

%@ @item navigate_assoc(@var{KeyList},@var{Model},@var{Assoc},@var{Result})
%@
%@ Navigates @var{Assoc} according to the list of navigation steps given in @var{KeyList}.
%@ This version return the found node as a Key-Value pair.
navigate_assoc0([Key],_,Assoc,Key-Element):-
        get_assoc(Key,Assoc,Element).

navigate_assoc0([refs,ClassKey|Rest],Model,Assoc,Result):-
        get_assoc(refs,Assoc,Refs),
        get_assoc(ClassKey,Refs,Object),
        (is_assoc(Object) ->
                (is_multiplicity_one_assoc(Object) -> get_first_key_value_assoc(Object,Key,
                    Element))
                ;
                get_assoc(ClassKey,Model,ClassList),
                get_assoc(Object,ClassList,Element),
                Key = Object),
        (Rest == [] -> Result = Key-Element ; navigate_assoc0(Rest,Model,Element,Result)).

navigate_assoc0([Key|Rest],Model,Assoc,Result):-
        get_assoc(Key,Assoc,Element),
        navigate_assoc0(Rest,Model,Element,Result).
```

```
%@ @item navigate_list(@var{Type-Index List},@var{Model},@var{List},@var{Result})
%@
%@ Navigation predicate for lists. Uses the navigation list on form [Type1-Index1,...,Typen-
    Indexn]
%@ to navigate to an element.
navigate_list([Type-Index],_,List,Result):-
        nth1(Index,List,Type-Result).

navigate_list([refs-Index,_-ClassKeyIndex|Rest],Model,List,Result):-
        nth1(Index,List,_-Refs),
        nth1(ClassKeyIndex,Refs,Key-Object),
        (Key-Object = _-_-_ -> resolveReference_list(Key-Object,Model,Element),
                               (Rest == [] -> Result = Element
                                              ;
                                              navigate_list(Rest,Model,Element,Result))
                               ;
                               (Rest == [] -> Object = Result
                                             ;
                                              navigate_list(Rest,Model,Object,Result))).

navigate_list([_-Index|NextIndex],Model,List,Result):-
        nth1(Index,List,_-Element),
        navigate_list(NextIndex,Model,Element,Result).

resolveReference_list(_-ClassID-ObjectID,Model,Object):-
        nth1(ClassID,Model,ClassList),
        nth1(ObjectID,ClassList,Object),
        !.

/*
SELECT
*/

%@ @item select_assoc(@var{NavLeft},@var{Rel},@var{NavRight},@var{Assoc},@var{Model},@var{-
    NewAssoc})
%@
%@ Iterates over @var{Assoc}, doing the navigation steps in @var{Nav} on each node, and checks
     the found element as 'Found Rel Value'
%@ If true, adds the orginating node to output tree.
select_assoc(NavLeft,Rel,NavRight,Assoc,Model,NewAssoc):-
        select_assoc(NavLeft,Rel,NavRight,Model,Assoc,List,[]),
        list_to_assoc(List,NewAssoc).

select_assoc(Key-Value) --> [Key-Value].
select_assoc([]) --> [].
select_assoc(_,_,_,_,assoc) --> [],{!}.

select_assoc(NavLeft,Rel,NavRight,Model,assoc(Key,Value,L,R)) -->
        select_assoc(NavLeft,Rel,NavRight,Model,L),
        select_assoc(Out),
        select_assoc(NavLeft,Rel,NavRight,Model,R),
        {check_attribute_assoc(NavLeft,Rel,NavRight,Model,Value) -> Out = Key-Value ; Out =
            []}.


%@ @item select_list(@var{NavLeft},@var{Rel},@var{NavRight},@var{List},@var{Model},@var{
    NewList})
%@
%@ Returns the sublist of a list, where each element in the sublist succeeded the relation
    check on the elements
%@ found using the two lists of navigation steps
select_list(NavLeft,Rel,NavRight,List,Model,NewList):-
       include(check_attribute_list(NavLeft,Rel,NavRight,Model),List,NewList).

/*
COLLECT
```

```
*/

%@ @item collect_assoc(@var{Nav},@var{Assoc},@var{Model},@var{NewAssoc})
%@
%@ Collects the elements from Assoc resulting from doing the navigation steps in @var{Nav}
%@ on each node of @var{Assoc}, and builds a new tree from those elements. O(N log(N))
collect_assoc(Nav,Assoc,Model,NewAssoc):-
        map_assoc_to_list(navigate_assoc0(Nav,Model),Assoc,Result),
        list_to_assoc(Result,NewAssoc).

collect_list(NavigationsList,List,Model,Result):-
        maplist(navigate_list(NavigationsList,Model),List,Result),!.

/*
INCLUDES ALL
*/

%@ @item includes_all_assoc(@var{WholeAssoc},@var{PartAssoc})
%@
%@ Checks if all Key-Value pairs in @var{PartAssoc} exists in @var{WholeAssoc}
%@ O(M log(N)), where M is the size of @var{PartAssoc} and N is the size of @var{WholeAssoc}
includes_all_assoc(_,assoc):-!.
includes_all_assoc(WholeAssoc,assoc(Key,Value,L,R)):-
        (get_assoc(Key,WholeAssoc,Value2),Value #= Value2 ->
                includes_all_assoc(WholeAssoc,L),
                includes_all_assoc(WholeAssoc,R)).


%@ @item includes_all_list(@var{Whole},@var{Part})
%@
%@ Checks if the list 'Part' is a sublist of the list 'Whole'
%@ The lists may be equal.
includes_all(_,[]):-!.
includes_all(Whole,Part):-
        sort(Part,PartSorted),
        sort(Whole,WholeSorted),
        subseq0(WholeSorted,PartSorted),!.
/*
ITERATORS
*/

%@ item map_assoc_to_list(@var{+Pred},@var{+Assoc},@var{-List})
%@
%@ Iterates through an assoc tree applying Pred on the value of each node. The result is
    appended to @var{List}
map_assoc_to_list(Pred,Assoc,List):-
        map_assoc_to_list(Pred,Assoc,List,[]).

map_assoc_to_list(_,assoc) --> [],{!}.
map_assoc_to_list(Pred,assoc(_,Val,L,R)) -->
        map_assoc_to_list(Pred,L),
        [Out],
        map_assoc_to_list(Pred,R),
        {call(Pred,Val,Out),!}.

/*
UTILITY
*/

%@ @item size_assoc(@var{Assoc},@var{Size})
%@
%@ Calculates the size (number of nodes) in an association list
size_assoc(assoc,0).
size_assoc(assoc(_,_,L,R),Size):-
        size_assoc(L,LeftSize),
        size_assoc(R,RightSize),
        Size is LeftSize + RightSize + 1.
```

```
%@ @item constrain_attribute(@var{+NavLeft},@var{Rel},@var{+NavRight},@var{+Model},@var{+Assoc
    })
%@ @PLXindex {constrain_attribute/5 (ocl_assoc)}
%@ Constrains found attribute (using list of navigations @var{NavLeft}) using the given @var{
    Rel} relation
%@ and comparing with navigation @var{NavRight}. Alternatively, an integer can take the place
    of any navigation.
%@ Expects @var{Assoc} and @var{Model} to be proper assoc trees
check_attribute_assoc(NavLeft,Rel,NavRight,Model,AttributeAssoc):-
        (integer(NavRight) -> ValueRight = NavRight ; navigate_assoc(NavRight,Model,
            AttributeAssoc,ValueRight)),
        (integer(NavLeft) -> ValueLeft = NavLeft ; navigate_assoc(NavLeft,Model,AttributeAssoc
            ,ValueLeft)),
        call(Rel,ValueLeft,ValueRight),!.

check_attribute_list(NavLeft,Rel,NavRight,Model,List):-
        (integer(NavRight) -> ValueRight = NavRight ; navigate_list(NavRight,Model,List,
            ValueRight)),
        (integer(NavLeft) -> ValueLeft = NavLeft ; navigate_list(NavLeft,Model,List,ValueLeft)
            ),
        call(Rel,ValueLeft,ValueRight),!.

%@ @item is_pair_list(@var{List})
%@
%@ Checks if @var{List} is a list of key value pairs where key is ground
is_pair_list([Key-_]):-
        nonvar(Key),!.
is_pair_list([Key-_|Next]):-
        (nonvar(Key) -> is_pair_list(Next)).

%@ @item length0(@var{List},@var{Size})
%@
%@ Augments built in length/2 in the following way:
%@ Simply calls the length/2 predicate, except when List and Size is a variable. In that case,
%@ size is given the domain 0..sup.
length0(List,Size):-
        (var(List),
        var(Size),
                Size in 0..sup
                ,!;
                length(List,Size)).

%@ @item element0(@var(Index),@var{List},@var{Element})
%@
%@ Augments clpfd:element/3 in the following way:
%@ Simple calls the element/3 predicate, except when List is a variable. In that case,
%@ @var{Index} is given the domain 0..sup, and @var{Element} is given the domain inf..sup
element0(Index,List,Element):-
         (var(List) ->
                Index in 1..sup, %then
                Element in inf..sup, %--||--
                !;
                element(Index,List,Element)). %else

%@ @item domain_interval(@var{List},@var{Interval})
%@
%@ Assign domain to a list of variables, where Interval is on the form: {0,56,8} etc.
domain_interval(List,Interval):-
        (nonvar(List) -> maplist(check_domain(Interval),List),!
                        ;
                        !).

check_domain(Interval,Element):-
        Element in Interval,!.

is_multiplicity_one_assoc(assoc(_,_,assoc,assoc)).
```

```
get_first_key_value_assoc(assoc(Key,Value,_,_),Key,Value).
```

Bibliography

[1] K. Pohl, G. Böckle, and F. Van Der Linden, *Software product line engineering: foundations, principles, and techniques.* Springer, 2005.

[2] K. Schmid and M. Verlage, "The economic impact of product line adoption and evolution," *Software, IEEE*, vol. 19, no. 4, pp. 50–57, 2002.

[3] D. M. Weiss *et al.*, "Software product-line engineering: a family-based software development process," 1999.

[4] R. Behjati, T. Yue, L. Briand, and B. Selic, "Simpl: A product-line modelling methodology for families of integrated control systems," tech. rep., Simula, 2011.

[5] R. Behjati, *A model-based approach to the software configuration of integrated control systems.* PhD thesis, University of Oslo, 2012.

[6] C. Atkinson, *Component-based product line engineering with UML.* Pearson Education, 2002.

[7] R. Behjati, S. Nejati, A. Gotlieb, T. Yue, and L. Briand, "Guided interactive configuration of embedded software systems using constraint satisfaction over finite domains," Tech. Rep. 2012-05, Simula Research Laboratory, 2012.

[8] "Sicstus jasper documentation." `http://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_12.html`.

[9] J. Cabot and E. Teniente, "A metric for measuring the complexity of ocl expressions," in *Model Size Metrics Workshop co-located with MODELS*, vol. 6, p. 10, 2006.

[10] "Atlas transformation language." `http://www.eclipse.org/atl/`.

[11] C. B. Carlsson M., Ottosson G., "An open-ended finite domain constraint solver, proc. programming languages: Implementations, logics, and programs,"

[12] K. S. Bordens, *Research Design & Methods.* Tata McGraw-Hill Education, 2006.

[13] J. Arlow and I. Neustadt, *UML 2 and the unified process: practical object-oriented analysis and design.* Pearson Education, 2005.

[14] *OMG Unified Modeling Language (OMG UML), Infrastructure (2.4.1).*

[15] *MOF Core specification (2.4.1).*

[16] M. Clavel1, M. Egea1, and M. A. G. de Dios, "The eos component." http://www.bm1software.com/eos/index.html, 2008.

[17] *Information Technology - Object Management Group Object Constraint Language*, september.

[18] J. Cabot, R. Clarisó, and D. Riera, "Verification of uml/ocl class diagrams using constraint programming," in *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*, pp. 73–80, IEEE, 2008.

[19] "Graphviz - graph visualization software." http://www.graphviz.org/Home.php.

[20] "Overview of constraint programming languages/libraries." `http://en.wikipedia.org/wiki/Constraint_programming`.

[21] C. Bessiere, "Constraint propagation," *Foundations of Artificial Intelligence*, vol. 2, pp. 29–83, 2006.

[22] P. Van Hentenryck, H. Simonis, and M. Dincbas, "Constraint satisfaction using constraint logic programming," *Artificial intelligence*, vol. 58, no. 1, pp. 113–159, 1992.

[23] M. van Amstel, S. Bosems, I. Kurtev, and L. F. Pires, "Performance in model transformations: Experiments with atl and qvt," in *Theory and Practice of Model Transformations* (J. Cabot and E. Visser, eds.), Lecture Notes in Computer Science, pp. 198–212, Springer Berlin / Heidelberg, 2011.

[24] S. Bosems, "A performance analysis of model transformations and tools," Master's thesis, University of Twente, 2011.

[25] "Atl user guide." `http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language#Lazy_Rules`.

[26] "Atl online forums." `http://www.eclipse.org/forums/index.php/f/241/`.

[27] "Atl commercial support at obeo." `http://www.obeo.fr/pages/maintenance-and-support/lts/en`.

[28] M. Carlsson *et al.*, *SICStus Prolog User's Manual*.

[29] P. E. Black, "Binary search tree." http://www.nist.gov/dads/HTML/binarySearchTree.html.

[30] P. E. Black, "Avl tree." http://www.nist.gov/dads/HTML/avltree.html.

[31] "Nist/sematech e-handbook of statistical methods." http://www.itl.nist.gov/div898/handbook/.

[32] P. Dalgaard, *Introductory statistics with R*. Springer, 2008.

[33] H. Störrle, "A logical model query interface," *International Ws. Visual Languages and Logic (VLLâĂŹ09)*, vol. 510, pp. 18–36, 2009.

[34] V. Acretoaie and H. Störrle, "Mq-2: A tool for prolog-based model querying," in *Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications (ECMFA 2012)*, p. 328.

[35] H. Störrle, "A prolog-based approach to representing and querying software engineering models.," *VLL*, vol. 274, pp. 71–83, 2007.

[36] "A tool for the formal verification of uml/ocl models using constraint programming." `http://gres.uoc.edu/UMLtoCSP/`.

[37] J. Chimia-Opoka, M. Felderer, C. Lenz, and C. Lange, "Querying uml models using ocl and prolog: A performance study," in *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*, pp. 81–88, IEEE, 2008.

List of Figures

List of Tables